

EROP to Augmented Drools Translator

Adrian Delchev

Updated May 2020.

Abstract

In today's technology driven world the presence of computing in the business and e-commerce spheres is becoming ever more dominant. The demand for automation of certain tasks and business operations is what makes the need of electronic contracts a pressing matter. Unlike a conventional paper based contract, which is likely to be polluted by ambiguities and legal jargon, an electronic executable contract is a translation of the latter that can be enforced by and acted upon by a contract management system.

Even though that at a high level of abstraction a contract is simply a document that specifies how the signing parties are to behave in various situations, the translation of a traditional paper-based contract to its electronic equivalent has proven to be a time consuming and a challenging task. The need of a specification language that is able to capture the essence of a contract in simple terms and definitions has been an ongoing research topic for many years. That research led to the development of the EROP language. EROP, which stands for events, rights, obligations and prohibitions, is a contract specification language that captures the building blocks of a contract into sets of events, rights and prohibitions.

Declaration

“I declare that this document represents my own work except where otherwise stated”

Acknowledgements

I would like to thank my supervisor, Dr. Ellis Solaiman, for the advice and support provided during the course of this project.

Table of Contents

Contents

Table of Contents.....	5
1 Introduction	7
1.1 Document Map.....	8
1.2 Motivation and Context of the problem.....	10
1.3 Aims and Objectives	11
2. Background.....	13
2.1 Contract Specification Languages	13
2.1.1 DocLog	14
2.1.2 CONTRACT	15
2.1.3 Other Contract Specification Languages.....	15
2.1.4 EROP, Augmented Drools and the CCC	16
2.2 From EROP to Augmented Drools	18
2.2.1 Mapping to Java Beans.....	18
2.2.2 Drools to R2ML.....	19
2.3 Summary of research.	19
3 Developing a solution.....	20
3.1 Design and analysis.	21
3.1.1 Compiler Analysis.....	21
3.1.2 Parser Generators	25
3.1.3 Solution Architecture.....	26
3.1.4 Functional and Non-functional requirements.....	28
3.2 Implementation	30
3.2.1 ANTLR Grammar.....	30
3.2.2 The Rule Structure Classes.....	32
3.2.3 Lookup and Mapping.....	35
3.3 Testing.....	36
4. Results, Evaluation and Conclusion.....	38
4.1 Contract in EROP translation Case study.....	38
4.2 Evaluation.....	43

4.2.1 Outcome and Role player Constraints	43
4.2.2 Resetting Rop sets.....	44
4.2.3 Case study evaluation.....	45
4.2.3 Evaluation of Aims and Objectives.....	47
4.2.4 Evaluation of Functional and Non-functional Requirements	48
4.2.5 Evaluation of the Software Engineering aspects	49
4.2.6 Skills learned	50
4.2.7 Conclusion	51
4.2.8 Future Work	52
5 References	53
6 Appendixes.....	55
6.1 User Manual.....	55
6.2 Mapping FR tests.....	58
6.3 Full translation of the Contract presented in section 4.....	67
6.4 Original Contract from section 4 in Augmented Drools	71
6.5 Formal refined grammar of EROP	76

1 Introduction

In the fall of 2014 I began work on a rather ambitious final year project with little previous experience in the field but with a clear vision in mind of the desired outcome. Over the course of the year my understanding of the problem matured and alongside that – my approach and aims of the project themselves matured. The result of those changes is reflected in the final design of the translator and it represents what I believe is a beneficial piece of software that would enable EROP to move closer to a concrete language rather than remaining a conceptual one. Throughout this document I try to present a concise and condensed report even for individuals with little knowledge of the field of electronic contracts and translation methods.

To ameliorate the navigation of this document I provide a content map of the document below. Those with different interests can use the map to skip sections or jump to points of interest quickly and easily. For each section I provide a brief introduction of the content to be covered and the addressed areas.

I begin with exploring my motivation for the topic of my final year project and the move on to outlining my aims and objectives and how I focused my development around them. At the end of the chapter I make note of the changes in my process throughout the year and how that affected my approach.

1.1 Document Map

The document is divided into the following sections:

- **Section 1 – Scene setting**

Firstly I examine the origin of my interest and my motivations for the topic as my final year project. This is meant to provide a context to the field and an introduction to some specific topics discussed later in the document. I also present my initial aims and objectives; showing what my intentions were when I first began work on the project and exploring the path I chose for developing the solution.

- **Section 2 – Background**

Section 2 provides an introduction into the world of electronic contracts and their significance as well as recent development in the field and the origin of the CCC and the EROP Language. It also provides a summary of contract specification languages and how different ones compare to one another and provides more reasoning as to why a translator is needed as well as some examples of translation techniques.

- **Section 3 – Developing a solution**

After exploring different types of translation techniques and their application for the project at hand I present different projects and how they tackled the task of translation between languages. I also review fundamental parts of any language to language translation and examine the tools used for the development of the project and their inner working and architecture. Alongside that I present the incremental stages of the development process and how they led to the final design. Finally I present some of the test cases showcasing different parts of the language syntax and how they translate to AD.

- **Section 4 – Results, Evaluation and Conclusion**

In order to show the correctness of the translator I provide case study, showing the mapping from EROP to AD and any changes to the original definition of the language since its first introduction. I then consider some of the functionality that was made redundant by newer versions of AD and how the developed piece of software relates to the original aims

and objectives of the project. I also review the ease with which the produced translator can be implemented and used with the CCC.

In the final review of the project I summarize my findings in relation of my evaluation of the original aims and objectives and any amendments to them. I consider each aim individually and critically examine whether or not it has been achieved and how it contributed to the development of the overall solution. I also highlight some of the directions for possible future development and improvement of the translator and its integration with the CCC.

• **Appendix**

Throughout the document I try to omit as much of the low-level development specific details and technicalities. For the ones wishing to review some of the particular finer details of different aspects of the development and testing process I provide the following content.

List of appendixes:

User Manual

Test Cases

Full translation of a sample contract

Formal refined grammar ofEROP

1.2 Motivation and Context of the problem

During my work placement I worked for a company in which interactions with business clients were heavily based on contractual bases. The majority of client communication was based on establishing the parameters of the contract signed between the company and the client. Even after the initialization of a contract, operations on said contract continued to be the driving force for speedy problem resolution and good client-provider relations. I worked specifically on a piece of software called CMS, short for Contract Management System. Its purpose was to keep the contracts up to date with information regarding the actions of the signing parties. The system dealt with almost every aspect of the client-provider cycle – Are the agreed upon goods delivered, is the financial part of the contract complied by, are there external consequences affecting the ability to provide goods or services. The amount of information stored in a contract absolutely astounded me but even so – I couldn't help notice that all that information had to be manually monitored and edited if need be. That was the beginning of my interest in contract management systems and the automation of contract compliance monitoring.

The demand for innovative ways of automating the process of contract compliance monitoring is coming both from the industry as well as the academia [1]. A well designed contract management system can reduce the manual aspect of contract compliance monitoring greatly and ensure that business processes of partners comply with the contract being enforced. Making sure that business operations and processes between the signing parties of a contract are carried out correctly and actually stipulate by the contract itself requires a management system to monitor the interactions. Such service called the CCC (Contract Compliance Checker) has been developed as an independent, third party monitoring service by researchers [2] to address the issue of contractual monitoring. The system itself was designed with the conceptual language called EROP in mind. EROP stands for events, rights, obligations and prohibitions and is a contract specification language that relies on the JBoss Rules[3], commonly known as Drools , for rule management.

The implementation of the CCC uses the EROP ontology - a set of the concepts and relationships within the domain of business to business interaction used for modeling the execution of business operations between partners and reasoning about the compliance of their actions. The EROP ontology is implemented in JAVA as an extension to the Drools engine, which allows for a better, more direct mapping of the EROP specification language to the concrete implementation. Having the option to express a contract in the EROP language allows for a broader user base since only a limited technical knowledge will be required to convey a contract in EROP as opposed to writing it in the extended Drools directly for monitoring. As it

currently is, a translation from EROP to the extended version of Drools (also known as Augmented Drools or AD) is a manual process, which requires the contract specification in EROP to be mapped to its AD equivalent. The automation of the process will contribute to the completeness of the paper to electronic contract translation process and allow for immediate use in the CCC. A translation engine for EROP to AD mapping will eliminate the need of manual translation, potentially eliminating any translation and mapping related errors and better utilization of the time spent expressing a paper contract as its electronic equivalent.

1.3 Aims and Objectives

The overall aim and end goal of the project is to design and implement a translation engine that would automate the process of EROP to AD translation and provides a valid and correct output for an input that complies with the specifications of the EROP language. I will measure the effectiveness of my solution by comparing already confirmed translations from the original papers that introduced the EROP language as well as comparing hand written and manual translations. In order to determine the extent to which the developed tool is successful I will evaluate the following objectives:

- **Iteratively develop a solution using a modular programming approach and development of unit tests.**

The resulting solution should be designed with a module structure in mind encapsulating different functionality and allowing for easier maintenance and enhancement in the future if needed. A clear architectural design and diagrams should be provided showing the inner workings at high level of abstraction and explaining the design decisions that led to the final solution. Example modules to be contained within the solution I/O, parsing, mapping, translation modules.

- **Define test cases that show different syntax and logic parts of the language**

The test cases used to evaluate the effectiveness of the tool should be selected with clear goal in mind and show the variety of the language concepts and constructs present in the EROP language and how the translation process effectively transforms them in their respective AD counterpart. The translation methodology should be made available as well as the formal grammar of the EROP language including a detailed analysis and reasoning behind any amendments, modifications and removals of the original version.

- **Research current translation methods and strategies and the use they find in the development of the tool.**

A big part of the development of the tool will be looking into existing translation tools and their effectiveness. The ideal solution will use various different sources for the final implementation picking and choosing the best practices and applications of the translation techniques, while noting the strengths and weaknesses of the different approaches in a written form. Any external packages and libraries used for the development of the tool should be noted and a high level overview of their workings and how they contribute towards the effectiveness of the solution should be given. Common translation methods should also be explored

2. Background

The presence of contracts in our society is ubiquitous – from every day simple oral agreements that we take for granted to formally specified and notarized documents that have strong and profound effect on our lives. Their wide use today is undeniable and its roots can be traced back to ancient societies [4]. The advancement of electronic commerce has increased the sheer number of contracts an organization can take part in, resulting in difficulties in keeping up with the requirements of an electronic market. Creating a contract is a task that requires significant resources and efforts – from hiring adept personnel to formally specify and verify the contract parameters to negotiation between parties and mediation with the business. Electronic contracting aims to automate the process of contract establishment and execution while reducing development costs and in order to achieve that a contract has to be captured in a language that has the expressive power to specify all the contract's content while eliminating any ambiguities[3].

2.1 Contract Specification Languages

In order for a contract to be eligible to be monitored and enforced by a contract compliance system of any kind it has to be formally specified in a language that has the ability to capture the requirements of a contract including legal requirements, clauses and internal policies as well as the acting parties and their actions. Given that the desired outcome of a contract monitoring and compliance service is automation and execution of contracts with minimal human interference, the language that captures the contract has to be precise and free of ambiguities so that the need of manual conflict resolution does not arise.

2.1.1 DocLog

In [5] the authors discuss the implications of e-commerce on the life of different entities without existence of a proper legal framework that is capable of regulating the behavior in terms of rights and obligations of users. It highlights that usually electronic message standards focus on practicality with the aim of ease of processing by back-office systems instead of human readability, which in turn leads to the omission of important elements of the purchase order, namely the General Terms and Conditions. Another drawback of message exchange based contract standards is the detachment of messages from meaning and consequences as well as concepts such as obligations, permissions and prohibitions.

The proposed contract representation language called DocLog aims to extend existing message standards, adding natural text to the messages and thus allowing negotiation and interpretation by human users. It also introduces the concept of legal advice system that is capable of providing, while not as sophisticated as an actual experienced law professional, legal advice about exchanged messages. DocLog uses a tri-layer structure that combines data, text and semantic oriented approaches for exchanging contract terms. The data layer aims to provide the contract data in such a way that it can easily and efficiently be processed by a transaction processing system [6]. The text approach is represented by a Natural language layer that strives to present the contract terms in a way that is easily comprehensible by human users. It uses XML to structure the content of a contract which allows the support for individual clauses, sub clauses, sections, sub sections and allows the use of version and approval management systems.

While DocLog does provide a relatively human readable way of capturing contract specifications, it doesn't provide any means of monitoring the captured contract or allow any manipulations on the captured properties of the contract. Furthermore several important aspects of contract specifications cannot be captured using the DocLog language. Temporal aspects such as – “Buyer has to submit payment no later than 5 days after making an order” as well as exceptional circumstances within contracts that specify what is to be done when an aberration from the norm of contract occurs are not presented in the original version of DocLog. Nevertheless the underlying architecture of the language provides insights on communication between different layers of the language and mapping between layers using EDI Translator [7].

2.1.2 CONTRACT

The CONTRACT Project [8] aims to develop a well-defined conceptual framework for contract based systems to which application entities can be mapped as well as to support management of contracts throughout any of the stages of a contract' life-cycle.

The presented administrative architecture [9] supports the management of e-contracts and defines the ontology used in a business partnership for the underpinned contract. This is covered in four steps:

- Consistency based off-line verification and achievability of contract aims given the possible reachable system states.
- Definition and compilation of the application specific processes that facilitate the execution of the contracts such as – enactment, monitoring, updating, termination, renewal etc.
- Definition of the roles of the interaction agents.
- Identifying different components and services used by acting agents necessary for them to play their respective roles in the partnership.

The main focus of the framework as discussed in [10] is centered around corrective monitoring , where violation of contractual norms are detected and then tried to be fixed using corrective measures as opposed to predictive monitoring where such violations are predicted by using the agent's behavior and actions are taken in order to completely avoid undesired behavior. The actual contracts are captured in an XML based language with a multi-layer architecture, whose inner working and capabilities are discussed in detail in [11].

The language is capable of expressing different contract structures such as clauses, parties, groups and actions using deontic notions such as obligations, permissions and prohibitions and even though it uses a relatively high level declarative style of writing its implementability is unclear. It is possible to represent concepts, intuitively acceptable to humans but unfortunately it is not possible to unambiguously translate them in a form that would allow it to be processed by machines. Unlike DocLog, the language offers a degree of exception handling; however it appears to be limited and is not the main focus of the project and neither is usability.

2.1.3 Other Contract Specification Languages

There are myriad other options when it comes to contract specification but unfortunately most of the available languages either do not focus on usability and practicality or don't have a contract monitoring system in which they can be integrated with ease.

- Rule ML [12] is a semantically neutral language for representation of rules but unfortunately significant parts of the translation between natural language contract to RuleML and then a language that can be formally verified cannot be automated.
- BPEL [13] uses event calculus to represent actions and their corresponding effects and is not targeted towards business contracts but for specifying web service interactions. It offers a less declarative approach than the other presented languages. Exception handling is present but recovery from exceptional cases is limited the biggest issue seems to be usability. The abstract notation may be too daunting for non-technical personnel to use for commercial purposes.
- Heimdahl [14] is a platform for monitoring obligation policies and it uses xSPL [15] for the specification of those policies. xSPL' syntax is declarative but some of the language constructs are not intuitive and might be difficult to understand for people without a technical background. Exception handling also doesn't appear to be possible directly.

2.1.4 EROP, Augmented Drools and the CCC

As introduced in [3] EROP is a contract specification language that focuses on execution and business resolution of a business partnership. The language relies heavily on events, rights and obligations, and it captures the information of a conventional paper-based contract into sets of the fore-mentioned constructs. One of the most significant additions that distinguish EROP from other contract specification languages is the extended capabilities that allow for reasoning and providing resolution of unforeseen circumstances that arise from business and technical failures. One of the strengths of the language stem from the fact that at its level of operation the low level details are abstracted away, allowing contract writers to concentrate on expressing the business operations of a contract. Another selling point of the language is its ease of use even for non-technical personnel and the already existing implementation of its ontology that allows for contractual compliance monitoring.

A contract written in EROP consists of two sections – a declaration section – where all the acting role players, business operations and composite obligations used in the rules are defined and a rule section that captures operations and manipulations of the entities specified in the

contract as well as any actions and exceptional behavior. The formal grammar of the language as well as more details on the EROP syntax will be provided in a later section.

The EROP ontology as specified in [3] is “a set of concepts and of their relationships within the domain of B2B interaction that we employ to model the evolution of interactions between business partners, for the purpose of reasoning about the compliance of their actions with their stated objectives in their agreements.” The ideas of the EROP ontology have been implemented as set of Java classes that capture the properties and available operations on those properties; the implementation extends the rule language offered by the Drools rule engine, adding various different construct to reason about and manipulate the operations of business partners within a contract. The implementation of the ontology, also known as Augmented Drools is less abstract and readable than EROP and closer to Java in style. It also needs additional code for convenience and housekeeping purposes that are needed for the implementation of the ontology to work but not necessary for human reader, initializing a contract in EROP.

The EROP language maps completely to Augmented Drools, which makes it possible for direct language to language mapping. In this sense, the problem of creating a precise and formal grammar of the EROP language becomes one of translating EROP to Augmented Drools, which is the main topic of this work and will be explored in detail in the upcoming sections.

The Contract Compliance Checker - for short the CCC is the contract compliance monitoring service as introduced in [2]. It is a neutral entity conceptually standing between the interacting parties and its purpose is to monitor the exchange of events between participating entities and infer whether or not the business operations these events relate to are compliant or non-compliant to a specified contract. The architecture of the CCC is illustrated in Fig. 1 but any further discussion on the inner workings of the CCC will be limited as it's not the topic of the work presented here. For the purposes of this project it is important to note that all contract expressed in EROP or Augmented Drools can be put into the CCC for contract compliance monitoring.

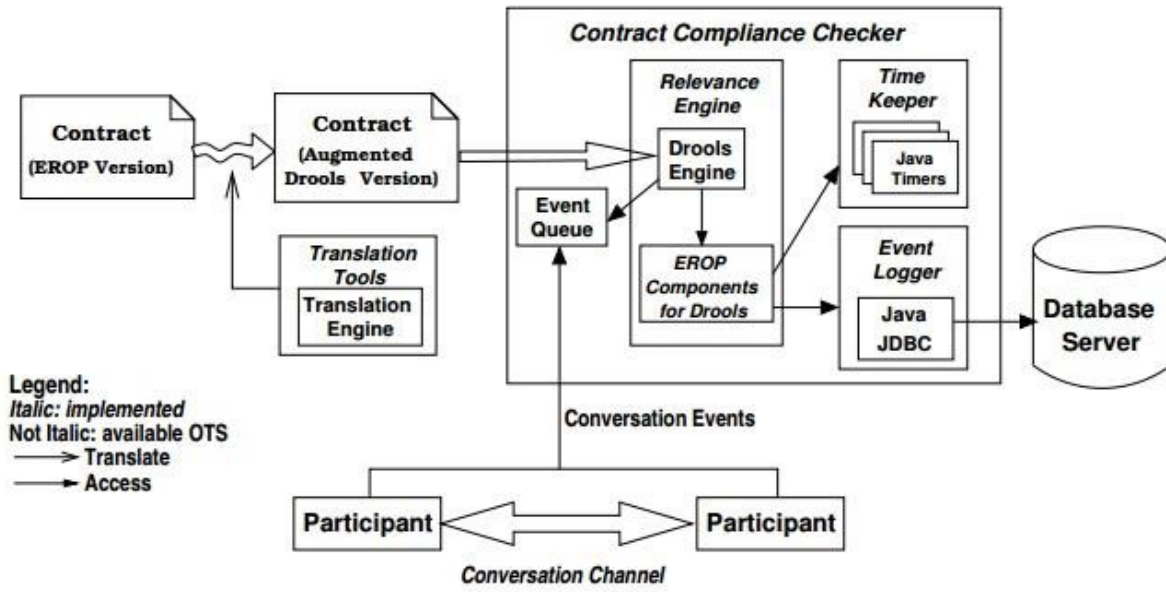


Fig1. Representation of the CCC architecture taken from [3]

2.2 From EROP to Augmented Drools

As mentioned in the previous section EROP maps directly to Augmented Drools given that the former is derived from the latter. In this case a translation between EROP to Augmented Drools essentially comes down to automating the mapping between the two languages. The translation techniques common for all most types of translations will be discussed and reviewed in detail in the next section, here I give some of the more specific techniques available.

2.2.1 Mapping to Java Beans

As discussed in [16], the translation of rules expressed in language based on natural or close to natural format to a rule standard can be accomplished by two-fold mapping where the extracted natural language is mapped to Java beans, which serve as intermediary for the

translation between the source language and JBoss Drools production rules. The translation technique uses a straight forward verb and noun concepts grammar that captures the parameters of the source language and is then mapped into rules.

Although the paper describes natural or close to natural language mapping, which requires more effort in comparison to EROP to AD mapping because of the lack of rule structure of the source language, it is interesting to note that an intermediary layer in the face of Java Beans is used in order to capture the required information to build an operational rule.

2.2.2 Drools to R2ML

The Authors of [17] discuss the approach used by Object Oriented rules systems such as Drools and ILOG Jrules. Such systems are built on top of Java vocabularies – in the case of Drools, Java beans are used as facts to represent the domain of the rules and their vocabulary in user applications. Different vocabularies are used by rules through the *import* declaration, specified inside of the rule file. The paper describes the approach used to translate from Drools using the low level structure of the language such as beans to translate rules to R2ML, which is an XML based Rule Markup Language.

Even though the direction of translation described in the paper goes in the opposite direction of the one desired from the EROP to AD translator, it does highlight the importance and benefits of using an XML structure or language for translation to Drools.

2.3 Summary of research.

In this section I covered the essential characteristics that a contract specification language should possess. I reviewed some of the most popular languages used to express a contract including their strengths, weaknesses and the ideas they are based on.

I proceeded to review EROP – the conceptual contract specification language, developed by researchers at Newcastle University. I highlighted what makes EROP stand out from other contract specification languages as well as the structure of the language and its concrete implementation in the face of the EROP ontology. I also did a cursory review of the CCC and the entire architecture of the system, showing how EROP and its ontology fit within it.

The conducted research demonstrates that among the presented contract specification languages, none has the expressive power or the ability to deal with exceptional circumstances arising from business or technical failures as EROP does. All of the presented languages require an extensive technical background and may be daunting for a non-technical person to use. In addition to that some of them are merely a notation for expressing contracts, with no definitive means of monitoring the expressed contract for compliance. EROP along with the CCC represents a complete solution for capturing contract requirements and then monitoring and enforcing them.

At the end of the section I briefly covered some specific translation techniques, without going into details about the common translation strategies as that is presented in a later section. A translator from EROP to AD has to be based on direct mapping and may make use of intermediary representations to hold its data. As presented in the reviewed approaches, JavaBeans-like structures and XML are capable of accomplishing the desired goal and have been used by other similar projects. The next section covers some of the fundamentals in translation and how they are applied in the development of the EROP to AD translator.

3 Developing a solution

The development of an EROP to Augmented Drools translator is a task that requires extensive analysis of current techniques and translation technologies. My initial aim was to approach the development process incrementally, dividing the overall project into several specific sub projects that would be united in the end to produce the final solution. During the initial stages of the development process the focus was on analysis and comprehension of the state of the art of the translation scene. The analysis revealed the parts of common translation techniques that would be essential for developing a translation and also gave an insight into the architectural design of the solution. The use of external, third parties libraries was taken into consideration and the necessity of such has been reviewed in greater detail later in this section.

The programming language used for the development of the solution is JAVA. As its implementation dependencies have been reduced to a minimum and it provides the required functionality to achieve the task at hand. The target of the translation – the implementation of the EROP ontology is also in JAVA, which makes the choice to use the language as a logical one

as it will promote consistency throughout the contract compliance monitoring solution [2] developed by the University researchers.

3.1 Design and analysis.

In my case the design and analysis stage has proved to be the lengthiest one during the development process. It included analysis of existing technologies and ideas and how they can be applied to the EROP to AD translator. During that stage I reviewed most common techniques in compilers and translators and the inner working of different technologies, trying to extract useful architectural designs patterns that can be applied to the task at hand.

3.1.1 Compiler Analysis.

Generally, the purpose of a programming language is two-fold [18] – they serve as a notation of describing computations to both machines and people. Other than formally expressing a programmer’s intention, they exist for the purpose of bridging the gap between different layers of abstractions – the higher layers that are easier to comprehend and safer to use and lower levels that are often times more efficient and flexible. But for a program to be run it needs not only to be expressed in a programming language- a language that is more human oriented , but it also needs to be translated to a language that a computer understands. Such translation software is known as a compiler.

A typical compiler breaks the mapping of a source language to the target language to several stages [19]. Most commonly the first of those is the analysis stage – it breaks a source program into pieces and verifies the grammatical structure of the source language on them. The resulting pieces are then used for the creation of an intermediary representation of the source language. The analysis stage’s duty is to detect syntactical and semantical compliance or inconsistency. The intermediate representation built by the analysis stage is called a symbol table and after its creating it is then passed to the next stage of the process. The second stage is the synthesis, where a translation to the target language is created using the intermediary representation of the source language. In a sense the analysis part is the front end and the synthesis is the back end of a compiler. When reviewed in more detail, the processes of a translation are executed in a sequence of steps (fig2.)

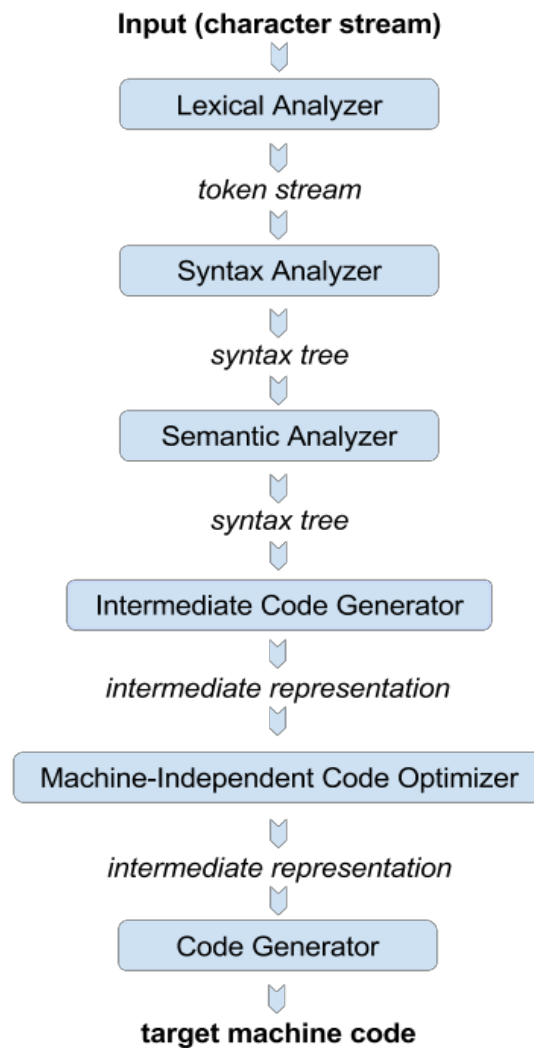


Fig2. Common phases of compiler translation inspired by [19]

The initial step of the translation process is carried out by the Lexical analyzer. Its purpose is to read the characters making up a source program or a file and use them to create meaningful sequences called lexemes, producing tokens containing the parsed information, which is then used in the syntax analysis stage. To put things into context an input in the form of the EROP language such as:

```
POAcceptance in buyer.rights
```

Would be broken down by a lexical analyzer into the following lexemes:

PAAcceptance is a lexeme that would be mapped to the token $(id, 1)$, where the 1 is pointing to the position of PAAcceptance in the generated symbol table containing information such as value and type.

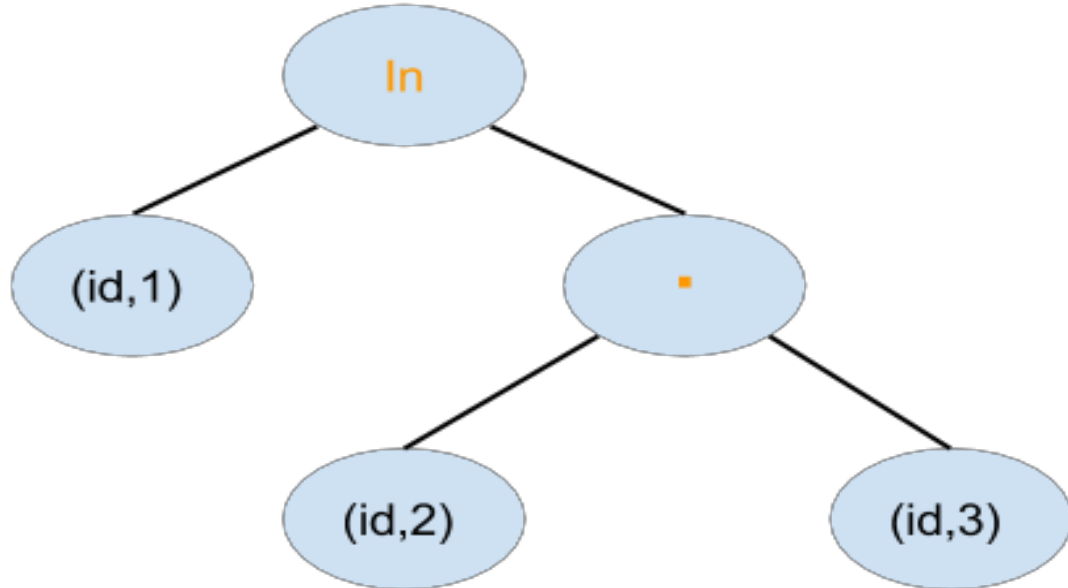
in and . Would be mapped to the tokens (in) and $(.)$ because they are both operations of the language

Similarly to POAcceptance, buyer and rights will be mapped to $(id, 2)$, $(id, 3)$

And the resulting token mapping would be:

```
(id,1) (in) (id,2) (.) (id,3)
```

The next step of a translation process within a compiled is the syntax analysis, also known as parsing. It uses the tokens created by the lexical analyzer to create a tree like intermediate representation of grammatical structure of the source language. Most commonly [19] the intermediary representation is known as a syntax tree in which each parent node represents an operation and the children nodes of the parent represent the arguments needed to complete the operation. Using the above breakdown of input characters into tokens, the resulting syntax tree for the given input would look like:



The node labelled as dot indicates that the operation must be completed using the children nodes and that the produced result should serve as the right hand side of the operation labelled in.

The next step of the process is the semantic analyzer. It uses the information stored in the symbol table as well as the generated by the syntax analyzer tree in order to check that the source program complies semantically with the definition of the language. One of the most important parts of the semantic analysis is type checking – in other words where the operands are from the appropriate type for the specified operator. For instance in most programming languages an array is indexed using an integer value, if the compiler detects anything that does not match the expected type it is supposed to notify the user for the inconsistency, in the case of EROP an example would be the definition of a business operation. As stated in [3] a business operation is defined by a generic string that starts with an upper letter. In that sense a string with a lower letter would be the wrong type when trying to define a business operation and therefore the compiler would have to return an error.

The step following the semantic analysis is the intermediate code generation. As stated in [19] the intermediary translation can be more than one or it can be expressed in a variety of different forms. The most important properties of the intermediate representation are that it should be easy to produce and it should be easy to translate into the target language.

The final step of the compiler's process is the code generation, which uses as an input the intermediate representation of the source program and maps that to the target language, where different approaches are used depending on the differences or similarities between the source and target languages.

The detailed view of a compiler's inner workings have been invaluable to the development and design process of the final solution. It has contributed towards the understanding of how typical language translators work and what the most common aspects in them are. Given the semantic similarities between EROP and Augmented Drools (the former is derived from the latter) a translation process between the two languages doesn't need to include all of the steps undertaken by a typical compiler, namely the code optimization step is redundant. Furthermore in order to speed up the development process and avoid needless low level errors oversights, specialized tools [19] that have efficiently implemented some of the outlined principles can be used.

3.1.2 Parser Generators

Parser generators are a specific class of software development tools that are able to generate the framework needed for a program to implement a parser from a set of rules called grammar. A few different methods exist for parsing a given stream of character input but the two most important ones are top-down and bottom up analysis algorithms [20] as they apply to the widest range of input grammars and context-free grammars and are appropriate to use in a parser generator.

Tools such as parser generators have been used in the creation on compilers and translators are their history can be traced back to the early days of computing with examples dating back to 1965 [21]. The two major advantages of using parser generators are firstly – development time – once proficient in writing grammars using a generated lexer and parser expedites the development process immensely. The second advantage of parser generators is correctness by construction, meaning that the generated parser accepts exactly the language specified in the grammar used to create it [21].

Today a wide range of parser generators exist, with all of them employing similar input parsing techniques and differing from one another in terms of style of grammar specification, algorithms used to parse the input, language of the generated parser files and so on.

As mentioned in the development of EROP, a parser generator called ANTLR was considered when thinking about the translation between EROP and Augmented Drools [3].

3.1.2.1 ANTLR

ANTLR is a parser generator that has a wide range of uses including reading, processing, executing or translating structured text or binary files. The latest version of ANTLR [22] – ANTLR4 support actions and attributes flexibility, meaning that different actions can be defined in separate files from the grammar and essentially decoupling it from the target language, enabling easier targeting of multiple languages.

ANTLR can also be used to generate tree parsers and processors of abstract syntax trees. It uses EBNF as a format of its grammar input and has support for popular IDEs. An additional benefit is that it generates a lexer as well as a parser and the resulting generated files are in JAVA format, which makes it consistent with AD and the language used for the development of the EROP to AD translator.

ANTLR is also widely popular and is used by Twitter for query parsing, processing over 2 billion queries a day as well as in projects such as Groovy, Hibernate, IntelliJ IDEA and many more [22]

3.1.3 Solution Architecture

After reviewing the most common translation techniques and approaches, the architectural design of the EROP to Augmented Drools translator started to emerge. The use of parser generator would enable a more rapid development and allow to some extent to reuse the formal grammar of EROP as specified in [3]. The use of ANTLR would also mean that the first three steps of typical compiler architecture can be accounted for and there is no need to create spate entities for the desired functionality. As noted earlier, machine independent code optimizer wouldn't be practical because of the similarities of the target and source languages, which essentially means that the intermediary code generator can interact directly with the code generator where the mapping is made and the final result is produced.

As discussed in previous section the accomplishment of an intermediary layer can be accomplished by an implementation similar to JavaBeans. The disadvantages of using JavaBeans directly are that it supplies nullary constructors for all of its subclasses, which means that they are at risk of being instantiated in an invalid state. The problem stems from the fact that a compiler cannot detect such instantiation which can lead to troublesome debugging and tracing, especially when using a generated parser. Nevertheless in order to accommodate the intermediary code generation layer of the EROP to Augmented Drools translator, similar in concept Collection of Java classes can be implemented that captures the essence of the building blocks of the EROP language. The collection of those classes, well as any additional classes required to accommodate communication between different layers of the architecture, will be discussed in the implementation section. The initial conceptual design of the translator is depicted in Figure 3.

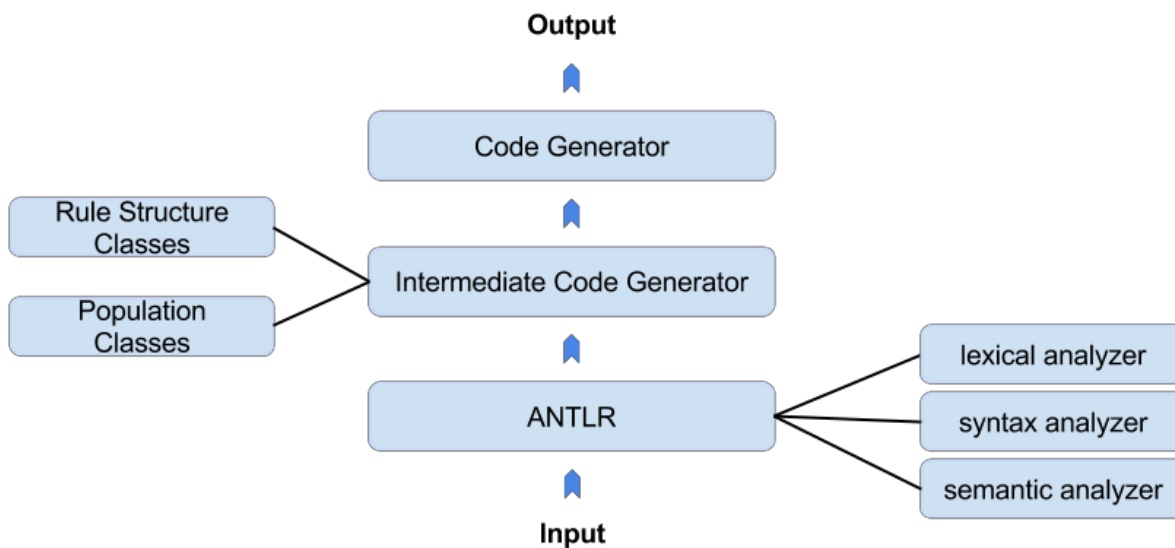


Figure 3. The Conceptual architectural design of an EROP to AD translator

3.1.4 Functional and Non-functional requirements.

Using the background research and analysis combined with the mapping examples provided in [3] the functional and non-functional requirements of the translator have been established. The requirements have been divided into two sections – General – capturing generalized requirements of the system as a whole and Mapping specific, capturing how the translator should handle details of the mapping process.

3.1.4.1 Functional Requirements

Mapping specific:

- FR1. The translator should always include the classes from the EROP Ontology.
- FR2. The translator should create any instances of the ontology classes used in the rule referencing.
- FR3. The translator should create ROP sets for every declared role player.
- FR4. The translator should maintain integrity of style conventions when translating names of business operations or composite obligations (capital in EROP but lowercase in AD)
- FR5. The translator should first translate the declaration section and only then the rules section.
- FR6. The translator should correctly translate keywords in EROP to the corresponding method calls in AD
- FR7. The translator should split rules in EROP that have an f-then-else section into two rules in AD.

General

- FR8. The translator should use a parser generator to parse an input file
- FR9. The translator should be able to create an intermediate representation of the parsed input
- FR10. The translator should be written in JAVA
- FR11. The translator should be of the form of a standalone GUI application, an executable script or both.

3.1.4.1 Non-Functional Requirements

General

- NFR1. The EROP input file should not be modified in the translation process.
- NFR2. The translator should work with the latest version of AD
- NFR3. A user manual should be created to show how the translator can be used.
- NFR4. The system should be well structured, encapsulating different functionality.

3.2 Implementation

The Implementation of the project has been an iterative process employing the agile software methodology and going back and forth between design, implementation and testing once the initial design modelling was finished. The length of the implementation sprints has been relatively short with sprints ranging from one to two weeks. This stage of the development of the solution involved an implementation of the grammar used for ANTLR as well as any java classes needed to make the translator operational.

3.2.1 ANTLR Grammar.

A grammar file in ANTLR is simply a file that specifies the syntax and different constructs of the language and how they connect with one another. On a high level of abstraction the grammar consists of lexer and parser rules that once specified are then embedded in the generated by ANTLR lexical and syntactic analyzers. The lexer rules begin with an upper case letter, as opposed to the parser rules and are used to tokenize the input. Lexer rules are essentially the fundamental, building blocks of a language. Parser rules on the other hand are more complex rules that can contain rules themselves as well as tokens characterized as fundamental to the Language.

As specified in [3], a contract expressed in the EROP language consists of two parts –a declaration section, where all the role players and business operations are defined and a rule section, where the different rules used for compliance monitoring are captured. From that we can infer that the root structure of the language is a contract file and everything else is contained within that file. That can be represented in ANTLR as the entry point of any received input and it would have any number of children depending on what a contract file can contain and what different constructs in the contract file can contain themselves. To put things into context, Fig4 gives a partial visual representation of what the structure of a contract file defined in the EROP language.

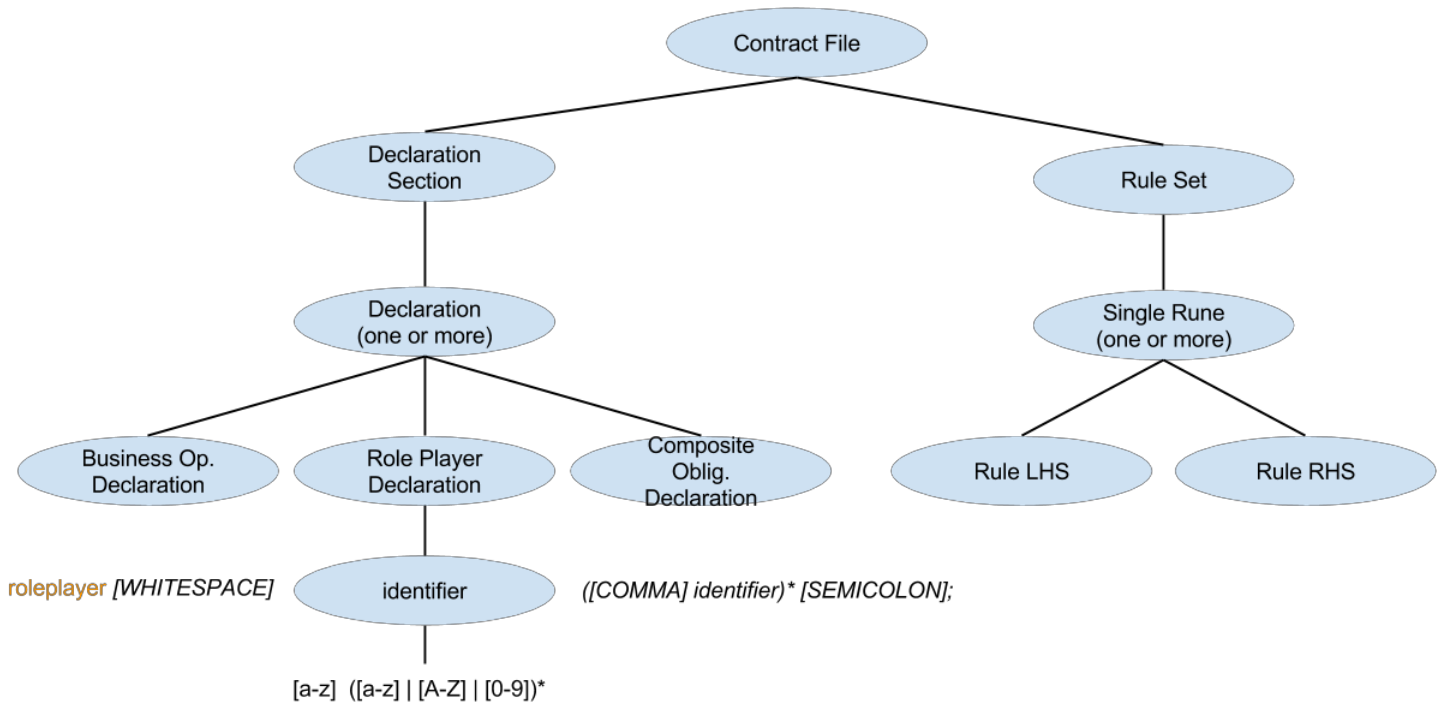


Figure 3. Partial representation of EROP grammar

As depicted in the partial representation, a declaration section can have one or more declarations and each declaration is a business operation, role player or composite obligation declaration. The role player declarations as well as the identifier are specified at the lowest level to give a feeling of what rules, sitting at the bottom of the rule hierarchy would look like. A role player declaration simply consists of the keyword ‘Roleplayer’ followed by whitespace and one or more identifiers and ending with a semicolon. The one or more quantifier makes it possible to declare multiple roleplayers in a single line as specified in [3]. An identifier is simply a string starting with lower case and the ability to contain uppercase letters as well as digits.

A grammar is an important part of ANTLR, but by itself it doesn’t provide much functionality because the associated parser is only able to tell us whether an input conforms to the language specification given. In order to build translation or any type of applications for that matter, there is a need for the parser to trigger some sort of action, whenever it encounters input sequences, phrases or tokens of interest. Fortunately ANTLR provides two mechanisms that allow invocation of actions – it automatically generates parse –tree listeners and visitors to enable building language applications [32]. A listener is an object that is able to respond whenever it detects rule entry and exit events triggered by a parse tree walker as it discovers and finishes nodes – which means that ANTLR automatically generates the interfaces for any entry or exit events. The most profound difference between listeners and visitors is that listener methods don’t have the obligation to explicitly call methods to walk their children – that gives flexibility in a scenario where only specific parts of the input language should trigger events.

The alternative is visitors – they must explicitly activate visits to their child nodes in order to keep the traversal of the tree going. In the case of an EROP to Augmented Drools translator the alternative method makes much more sense as parsing of all the input information is needed.

The provided functionality allows for triggering specific events when a rule from the grammar is entered. There is still the need for implementing specific actions when such events occur. Given that reusability is an important part of the software development process it would make sense to reuse common concepts instead of creating duplication. Identifier is an example of commonly repeating grammar structure that can be reused. It is used to describe business operations, composite obligations as well as roleplayers, not to mention that it can occur not only in the declaration section but also in the rule set when roleplayers and business operations are referenced or their ROP sets manipulated.

In order to allow reusability while keeping the ability to distinguish for which part of the grammar common grammatical structures refer to I've implemented two additional JAVA classes that serve as a buffer for the ANTLR parser and population classes that create the intermediary representation of the EROP language. Those classes are Variables Flagger and Variables memory. Their purpose is to respectively activate various different flags whenever the ANTLR tree walker enters different rules and then use those flags in order to make decisions on where the contents of the parsed file should be stored. The separation of communication between the ANTLR parser and the intermediary representation of EROP follows good software development guidelines and practices as it encapsulates and abstracts away the logic needed to make the decisions about where the parsed information goes. It also helps with testing and contributes to the modular approach design, which is one of the development aims of the project.

3.2.2 The Rule Structure Classes.

As discussed in the analysis section, the intermediate code generator of the translator can be accomplished by custom Java classes inspired by various different techniques. The resulting Java classes would have to capture the structure of the corresponding language constructs and any information they hold that are needed for linking the intermediary representation to the final target language.

The hierarchical structure of the classes corresponds to the implementation of the ANTLR grammar and is based on the language constructs of EROP as presented in [3] and the initial draft of the grammar provided in the fore-mentioned paper. The resulting classes and their functions are as follows:

- EventMatchCondition – represents the conditions an event match has to satisfy in order for the event to be triggered. It follows the structure field – operator – value and as specified in the full language grammar (included in the appendix) the field value can be any one from botype/outcome/originator/responder as specified in [3]
- Constraint – the constraint class is a generalized collection of any constraints that can be specified on a rule. It can hold any of the following :
 - RopConstraint – capturing the presence of absence of particular business operations or composite obligations in a role player’ ROP sets.
 - HistoricalConstraint – used to condition the triggering of rules depending on the presence or absence of certain events or the times a specified event occurred.
 - TimeDirect and TimePartialComparisons – constraints used to enforce additional checks on the timestamp of a given event. TimeDirectComparison is used to check of a timestamp the same as, before or after a specified point in time. The TimePartialComparison is used to check if an event timestamp is within a given range of hours, minutes, years, days or months.
 - OutcomeConstraint – used to specify a constraint on the outcome of an event such as Success, Fail, BizFail etc.
- RhsAction – represents the right hand side of a rule – anything between the ‘then’ and ‘end’ part of a rule. It can contain conditional statement, outcome or pass actions as well as any manipulation on a role player’s ROP sets or the outcomes of a business operation.
- IfStatement – a conditional structure that is used to capture additional constraints in the RHS of a rule. It comes with its own left and right hand side and even though it doesn’t alter the EROP language’s structure it allows for a more natural and productive style of writing.
- AddOrRemAction – used to gather information about any manipulation of a role player’ ROP sets such as adding or removing Rights/Obligations/Prohibitions.
- Rule – the root class in the rule class structure architecture that contains all the information required to represent and recreate a rule.

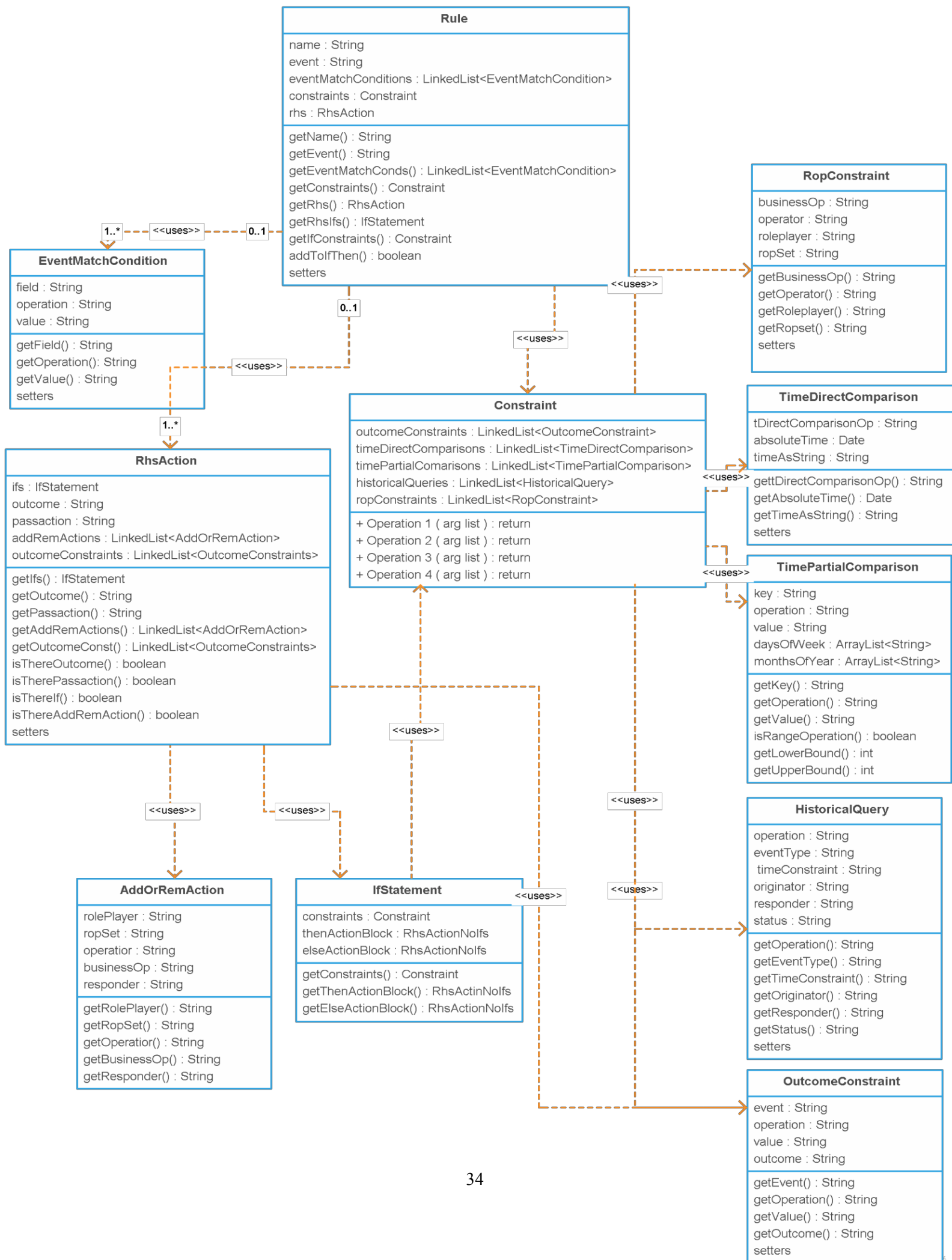


Figure 4 shows the architectural hierarchy of the classes in the Rule Structure and how they interact with one another.

3.2.3 Lookup and Mapping.

With a suitable ANTLR grammar and an intermediate Structure to represent EROP, the only thing left to make a working translator is a Mapper. The Mapper Functionality is executed by the Translator Java Class. Its purpose intuitively is to translate different parts of the rules from the intermediary format to the final target language – Augmented Drools. Given that the intermediary format is quite close to the target Language a direct mapping, with some decision making is a suitable option for the translation.

The Translator also includes decision making logic that allow it to determine whether a single rule in the source language needs to be broken down into multiple ones in Augmented Drools. This is most commonly due to conditional statements in the rule structure of the source language and is due to the fact that conditional statements do not exist in Augmented Drools. A quick example to illustrate the mapping process is given:

```
rule "Rule1"
  when e matches (eventMatchConds)
  then
    if (booleanConditions)
    then
      actionBlock1
    else
      actionBlock2
    end
  end

rule "Rule1IfThen"
  when $e: Event (eventMatchConds)
  eval (booleanConditions)
  then
    actionBlock1
  end

rule "Rule1IfElse"
  when $e: Event (eventMatchConds)
  eval (! booleanConditions)
  then
    actionBlock2
  end
```

In general a rule in EROP containing an if else statement maps to two rules in Augmented Drools – one with the if condition added to the when condition set in AD and the then action added to the right hand side of the AD rule. The second rule contains the negated if condition to the when condition set and the else action added to the right hand side of the AD rule. It's worth noting that the second rule only needs to be generated if there is an else action

block, in the case where there is only an if action block, the second rule doesn't need to be translated.

The final piece of the translator is the Lookup, implemented as a Java Class using an XML file containing the corresponding mappings of methods of the EROP Ontology classes to the keywords in EROP. The design is inspired by the various XML languages used as an intermediary layers as discussed in the previous sections and is also quite practical because it encapsulates all the mapping to a single file. In the case when the name of a method is changed in Augmented Drools, the only change needed to be made to the Translator is in the lookup file.

3.3 Testing

Testing played an integral part in the development of the project and, as mentioned earlier, has been interleaved with the incremental design and implementation. Unit tests have been developed to test the correctness of the classes of the rules structure and in fact were needed to determine the need for the buffer classes needed to help with the population of the rule structure classes.

Another, perhaps even more crucial part of the testing process was the development and testing of the ANTLR grammar. It was developed incrementally, which helped identify some of the drawbacks of the original grammar of the language as presented in [3]. That resulted in amendments and changes that will be presented in the next section along with a discussion.

A collection of test cases has been added to the appendix section in order to showcase different test scenarios and how the translator handles various different language constructs and their translation. Testing has also enabled the generation of results that would help in the evaluation of the project objectives. The changes made as a result of testing helped in the construction of the final architectural model of the project that as depicted in Fig 4.

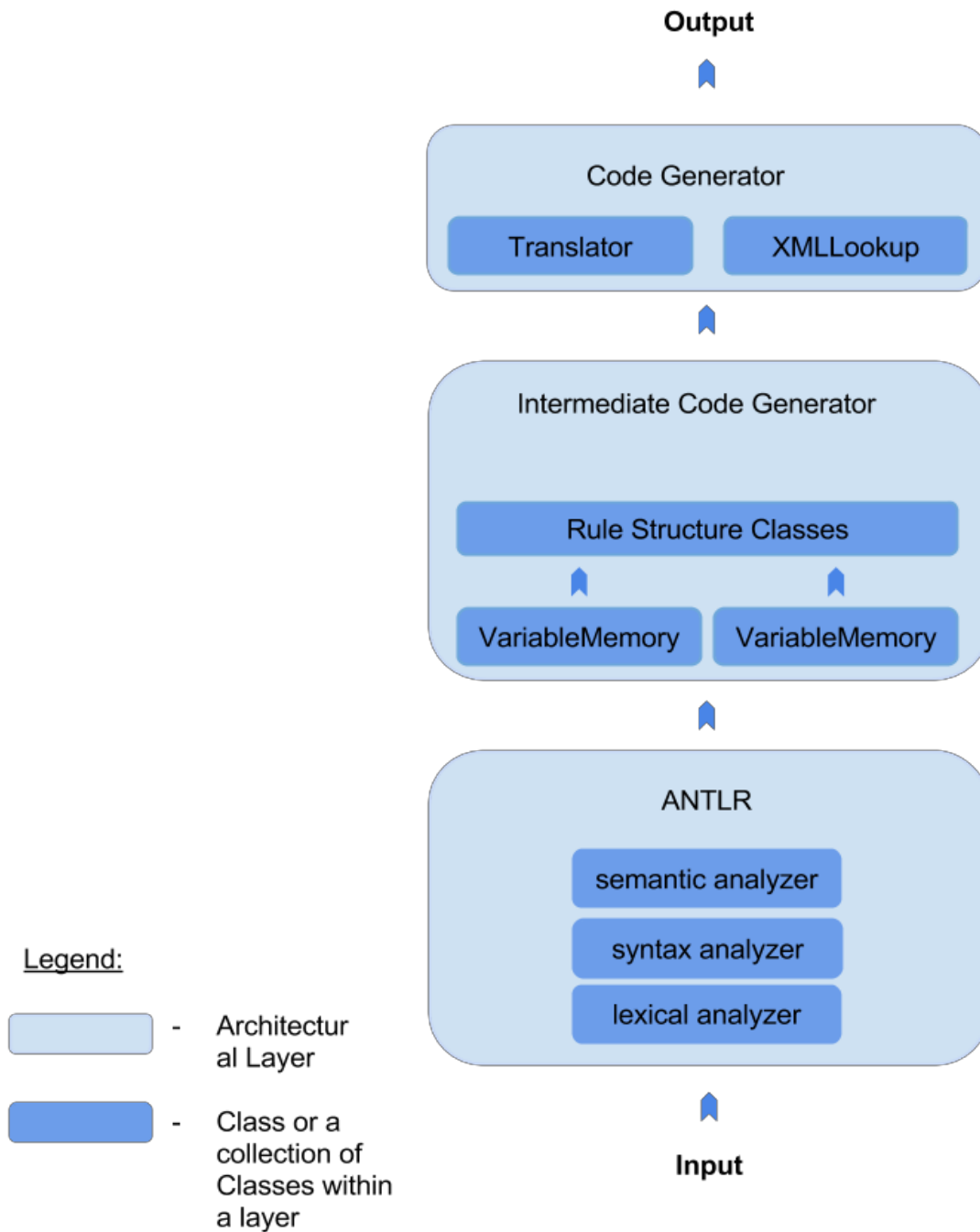


Figure 4. Final architecture of the project after a few rounds of incremental testing and development

4. Results, Evaluation and Conclusion

4.1 Contract in EROP translation Case study.

In order to critically examine and determine the correctness of the developed tool as well as any addition, enhancements and/or omissions that need to be made to both the EROP language or the Augmented Drools implementation, I present a case study in which the concepts of a contract represented in the latest version of Augmented Drools are extracted and serve as key points that are then used to express the same contract in EROP. Once the contract is in EROP it can then be ran through the translator and the produced translation can be evaluated against the original contract, expressed in Augmented Drools. The two main aspects on which the translation will be judged are ability to express concepts and correctness of the produced translation with regards to the original of the contract.

The contract used for the case study is between two parties, which will be referred to by simply *Buyer* and *Seller*. The original contract in its entirety, expressed in Augmented Drools can be found in the appendix section. The clauses of the contract extracted from the original are as follows:

- C1: The buyer has the right to submit a buy request, having send a buy request, a buyer's right so submit any further ones is revoked until the current one is resolved. At the same time, the seller gains an obligation to either accept or reject the received buy request.
- C2: In the event of one or more business failures during the buy request, the first business failure should be noted and any further business failures should reset the Rop sets of the role players.
- C3: Having received a rejection of a buy request, the pending obligation is satisfied and the buyer can have its right to send additional buy requests restored.
- C4: In the event of one or more business failures during the rejection of the buy request, the first business failure should be noted and any further business failures should reset the Rop sets of the role players.
- C5: After receiving an acceptance of a buy request from the seller, the pending obligation has been satisfied and the buyer receives a new obligation to pay the seller as well as the right to cancel the order.
- C6: In the event of one or more business failures during the acceptance of the buy request, the first business failure should be noted and any further business failures should reset the Rop sets of the role players.

- C7: After a payment has been received, the buyer has satisfied its obligation; he loses the obligation to pay as well as the right for a cancellation and regains his right to submit further buy requests.
- C8: In the event of one or more business failures during payment of the buy request, the first business failure should be noted and any further business failures should reset the Rop sets of the role players.
- C9: After the buyer sends a cancellation, he loses the obligation to pay and the right to submit further cancellations.
- C10: In the event of one or more business failures during cancellation of the buy request, the first business failure should be noted and any further business failures should reset the Rop sets of the role players.

Two role players are defined in the contract along with the following business operations: BuyRequest, Payment, BuyConfirm, BuyReject, and Cancelation. The clauses of the contract as specified above, define how the ROP sets of the roleplayers change during the course of the interaction.

A file in EROP starts with the definition of the role players, business operations and composite obligations used in the contract.

```
roleplayer buyer, seller;
businessoperation BuyRequest, Payment, BuyConfirm, BuyReject, Cancelation;
compoblig ReactToBuyRequest(BuyConfirm, BuyReject)
```

The second part of the contract contains definition of the rules of the specified roleplayers and interactions between them. The rule for a received Buy Request can be derived from C1 of the contract. It occurs when a successful buy request is received.

```
rule "BuyRequestReceived"
  when ematches (botype == BUYREQ, originator == buyer, responder ==
store, outcome == success)
    BuyRequest in buyer.rights
  then
    buyer.rights -= BuyRequest(seller)
    seller.obligs += ReactToBuyRequest(buyer, "01-01-2016 12:00:00")
end
```

The second rule can be derived by C2 and as specified it requires specific actions to be executed depending on a certain condition. The rule can be modelled using a conditional structure in EROP.

```
rule "BuyRequestBnessFailure"  
  when ematches (botype == BUYREQ, originator == buyer, responder ==  
store, outcome == tecFail)  
    BuyRequest in buyer.rights  
  then  
    if (BuyRequest.BizFail == false)  
      then BuyRequest.BizFail == true  
      else reset buyer  
        reset seller  
    endif  
end
```

The third rule of the contract is derived from C3 and is triggered whenever the seller rejects a buy request from the buyer.

```
rule "BuyRequestRejected"  
  when ematches (botype == BUYREJ, originator == store, responder ==  
buyer, outcome == success)  
    ReactToBuyRequest in seller.obligs  
  then  
    seller.obligs -= ReactToBuyRequest(buyer)  
end
```

The fourth rule of the contract is directly derived from C4 and is very similar in definition to BuyRequestBnessFailure. It occurs when a business failure occurs during a rejection of a buy request.

```
rule "BuyRequestRejectedFailures"  
  when ematches (botype == BUYREJ, originator == store, responder ==  
buyer, outcome == tecFail)  
    ReactToBuyRequest in seller.obligs  
  then  
    if (BuyConfirm.BizFail == false)  
      then BuyConfirm.BizFail == true  
      else reset buyer  
        reset seller  
    endif  
end
```


The fifth rule defines what happens when a successful confirmation of the buy request is received, it is derived from C5.

```
rule "BuyRequestConfirmation"
  when e matches (botype == BUYCONF,originator == seller,responder ==
buyer,outcome == success)
    ReactToBuyRequest in buyer.obligs
  then
    seller.obligs -= ReactToBuyRequest(buyer)
    buyer.obligs -= Payment(seller)
    buyer.rights -= Cancellation(seller)
end
```

The sixth rule, similarly to the second and fourth rules describes what happens in the event of failures during the confirmation.

```
rule "BuyRequestConfirmationFailure"
  when e matches (botype == BUYCONF,originator == seller,responder ==
buyer,outcome == tecFail)
    ReactToBuyRequest in seller.obligs
  then
    if (BuyConfirm.BizFail == false)
      then BuyConfirm.BizFail == true
      else reset buyer
      reset seller
    endif
end
```

The seventh rule, derived from C7, captures what occurs in the event of a successful payment.

```
rule "PaymentReceived"
  when e matches (botype == BUYPAY,originator == buyer,responder ==
store,outcome == success)
    Payment in buyer.obligs
  then
    buyer.obligs -= Payment(seller)
    buyer.rights -= Cancellation(seller)
end
```

The eighth rule describes what happens in the event of exceptional circumstances during receiving a payment, it is derived from C8.

```

rule "PaymentReceivedBFailures"
  when ematches (botype == BUYPAY, originator == buyer, responder ==
store, outcome == tecFail)
    Payment in buyer.obligs
  then
    if (Payment.BizFail == false)
      then Payment.BizFail == true
      else reset buyer
      reset seller
    endif
  end
end

```

The ninth rule captures what happens whenever a cancelation is received.

```

rule "BuyCancellation"
  when ematches (botype == BUYCANC, originator == buyer, responder ==
store, outcome == success)
    Cancelation in buyer.rights
  then
    buyer.rights -= Cancellation(seller)
    buyer.obligs -= Payment(seller)
  end
end

```

The last rule of the contract expresses what is to happen whenever exceptional circumstances occur during the a buy cancellation

```

rule "CancellationBFailures"
  when ematches (botype == BUYCANC, originator == buyer, responder ==
store, outcome == tecFail)
    Cancellation in buyer.rights
  then
    if (Cancelation.BizFail == false)
      then Cancelation.BizFail == true
      else buyer reset
      seller reset
    endif
  end
end

```

4.2 Evaluation

4.2.1 Outcome and Role player Constraints

I will start by discussing the ability of EROP to express concepts present in the latest version of Augmented Drools as seen in the original of the contract presented in the appendix section. Clauses 2, 4, 6, 8 and 10 of the contract require the ability to check if a certain business action has been set as a business failure, this can be characterized as an outcome constraint. In the original version of EROP outcome constraints exist, but they are targeted at the event match conditions. For example, in the original specification of EROP the following syntax was possible:

```
rule "Sample"  
    when e matches (botype == BUYCANC, originator == buyer, responder ==  
store)  
        e.outcome == success  
    then  
  
end
```

Where e is the event name and outcome is a property of the event. This made it possible to omit certain fields in the event match condition block and specify additional constraints after it. In the latest version of the implementation of Augmented Drools, an event match block requires all of the fields (type/originator/responder/status) to be specified. This defeats the purpose of the Outcome constraints as introduced in the original version of EROP and makes it so that it is no longer needed in the form that it was introduced; however – in the latest version of the ontology, as seen in the contract, outcome constraints can be used on business operations to check for example if the business operation has failed. This is expressed in clauses 2,4,6,8 and 10 of the contract presented in the previous section. The functionality to make such checks were not present in the original version of EROP, to accommodate it I've amended the original outcome constraint to have the following syntax and role.

```
BusinessOperation.BizzFail == true/false
```

The construct can be used both in the Left hand side and the right hand side of a rule, with the same syntax but a different meaning. When used in the left hand side it is placed after the event match condition, the same way as it was introduced originally. The role when placed in the left hand side of a rule is to check if the specified business action has happened, in other words it is a Boolean condition.

When used in the right hand side of a rule – it serves not as a Boolean condition but rather a way to specify that the condition happened or didn't happen. In other words when used in the right hand side of a rule it serves as a setter. The change makes possible expressing clauses 2,4,6,8 and 10 and it updates the no longer needed version of outcome constraints as expressed in the original version of EROP in [3]

The requirement of the latest version of Augmented Drools that all of the event match condition fields have to be specified also makes the Roleplayer constraint obsolete. The following syntax is no longer needed and can be removed from the language.

```
rule "Sample"
  when e matches (botype == BUYCANC)
    e.originator == buyer
    e.responder == store
  then
end
```

The same as outcome constraints, role player constraints were used to add additional Boolean conditions after the event match block, given that some of the event match condition fields were omitted, given that the methods that were used to check for that functionality have been removed from the implementation of Augmented Drools it is no longer possible to do that.

4.2.2 Resetting Rop sets.

The contract in Augmented Drools has another feature that is not present in the original specification of the EROP language. It is also captured by clauses 2,4,6,8 and 10 and it gives the ability to reset the ROP set of a given role player. This is needed to keep the consistency of the ROP sets of roleplayers in the case of certain exceptional situations such as technical or business failures. To accommodate that functionality I've added the keyword *reset* to the grammar of EROP, which enables the contract writer to reset the ROP sets of a given roleplayer. It can only be used in the right hand side of a rule, similarly to ROP set manipulation. A sample of the operation is as follows:

```
rule "CancellationBFailures"
  when e matches (botype == BUYCANC, originator == buyer, responder ==
store, outcome == tecFail)
  then
    seller reset
end
```

4.2.3 Case study evaluation.

When the contract, shown in the previous section is inputted in the Translator it produces the following results. A rule file in Augmented Drools, like one in EROP, starts with a declaration section where all the objects and entities used in the file are declared. After some java statements to import the classes of the EROP ontology, there is a section to declare global identifiers such as Role Players, Composite Obligations and Business Operations. Augmented Drools also needs instances of some other EROP ontology classes such as the Relevance Engine and the Event Logger for reference in the rules. The translator also automatically generates ROP sets for each Role Player specified in the declaration section (conforming by functional requirements FR1, FR2, FR3 and FR4). The translated declaration section looks as follows:

```
package BuyerStoreContractEx
import uk.ac.ncl.erop.*;
import uk.ac.ncl.logging.CCCLogger;

global RelevanceEngine engine;
global EventLogger logger;

global RolePlayer buyer;
global ROPSet ropBuyer
global RolePlayer seller;
global ROPSet ropSeller
global BusinessOperation buyRequest;
global BusinessOperation payment;
global BusinessOperation buyConfirm;
global BusinessOperation buyReject;
global BusinessOperation cancelation;
```

The translator correctly generates instances of the Relevance Engine and Event Logger as well as the two Role Players and their corresponding ROP sets and all the specified Business Operations (Operations names start with lower case due to the fact that they are java object and must follow Java style rules, as specified in FR5).

The syntax to define rules in Augmented Drools is the same as in EROP given that the latter is derived from the former and has the following structure:

```
rule RuleName
    when conditions
    then actions
end
```

The Translator produces the following translation of the first two Rules:

```

rule "BuyRequestReceived"
  when $e: Event (type=="BUYREQ", originator=="buyer", responder=="store",
status=="success")
    eval (ropBuyer.matchesRights (buyRequest) )
  then
    ropBuyer.removeRight (buyRequest, seller);
    BusinessOperation[] bos = {buyConfirm, buyReject};
    ropSeller.addObligation("ReactToBuyRequest", bos, buyer,"01-01-
2016 12:00:00");
end

```

The placeholder event variable is correctly translated to $\$e$ and the event match conditions are specified in the Augmented Drools format. Constraints on event attributes is imposed outside the event match using the *eval* keyword as well as the methods from the Augmented Drools implementation (as specified in FR6). The right hand side of the rule is translated correctly – with manipulation of ROP sets going through method calls of the generated ROP sets of roleplayers. As expected, in the case of composite obligations, an extra line of code is needed to add a new composite obligation. In the above translation a composite obligation called *bos* is created and it consists of two other business operations.

The second EROP rule, derived from clause 2 is translated to two rules in Augmented Drools because of the conditional structure used.

```

rule "BuyRequestBnessFailureIfThen"
  when $e: Event (type == "buyreq", originator == "buyer", responder ==
"store", status == "tecfail")
    eval (buyRequest.getBusinessFailure() == false)
    eval (ropBuyer.matchesRights (BuyRequest) )
  then
    buyRequest.setBusinessFailure == (true)
end

rule "BuyRequestBness1stFailureIfElse"
  when $e: Event (type == "buyreq", originator == "buyer", responder ==
"store", status == "tecfail")
    eval (!buyRequest.getBusinessFailure() == false)
    eval (ropBuyer.matchesRights (BuyRequest) )
  then
    ropBuyer.reset ();
    ropSeller.reset ();
end

```

The single rule in EROP is correctly broken down into two rules in Augmented Drools (as specified in FR7) – the first one consisting of the conditions of the if statement added to the left hand side of the rule and the then action added to the right hand side of the rule. This rule also demonstrates the changed outcome constraints at work, correctly matching them as Boolean conditions and setters on the appropriate places (Lhs/Rhs). The second rule is produced by adding the negated conditions of the if statement to the left hand side of the rule, while adding the then action to the right hand side of the rule. It also demonstrates the translation of the newly added reset construct that allows a contract writer to reset the ROP sets of a given role player.

The rest of the translation produces similar results, correct for the constructs used. The full translation is attached in the appendix section and can be compared against the original of the contract, which is also included, to verify its correctness.

4.2.3 Evaluation of Aims and Objectives.

When I started capturing the aims and objectives of the project back in November, I specified three main aims:

- Iteratively develop a solution using a modular programming approach and development of unit tests.

While the development of the final solution was accomplished in an incremental fashion and the architectural structure of the solution can be specified as modular, the part of the objective that I don't think I've accomplished fully is the development of unit tests. This is largely due to the fact that at the time of specification of the original aims I didn't exactly know where the development would take me and what would be required. Having completed the project I can say that unit testing wouldn't be the perfect testing strategy in the development of the translator, because the testing requirements were to a greater extent related to the development of the right grammar for ANTLR and then testing that grammar with various different language constructs to ensure that the grammar does in fact allow them to be expressed and parsed correctly. In that sense I've changed the testing strategy and that can be seen in the appendix related to testing. The aim has largely been met with some minor room for improvement with regards to the better definition of the test cases at the beginning of the project.

- Define test cases that show different syntax and logic parts of the language

I believe I've met this aim of the project as shown in the different test cases attached in the appendix section as well as the reviewed Case study in the previous section. Testing the different language constructs not only showed the correctness of the translator but it was also vital in the discovery of obsolete language structures and the need for new ones, as shown in the case study. In my opinion the aim was met fully and is perhaps one of the most important parts of the project aside from the development of the translator itself.

- Research current translation methods and strategies and the use they find in the development of the tool.

Research was perhaps one of the most crucial parts of the project, consuming a significant portion of the project's allocated time. It wasn't confined to just researching translation itself but the project also required an extensive background research on electronic contracts, different contract specification languages and the university research leading to the development of EROP, the Contract compliance checker and Augmented Drools. The translation of different translation techniques, the structure and inner workings of compilers in particular was crucial to the development of the project. The different reviewed papers with translation techniques to different rules languages were also instrumental for the development of the final solution. Every translation aspect researched in the background section and especially in the analysis section found use and if not directly, then at least served as an inspiration in the development of final architectural design and implementation of the project. The objective has been met and was the most important one in regards to learning new material and software development techniques as a whole.

4.2.4 Evaluation of Functional and Non-functional Requirements

The entirety of the Mapping specific functional requirements, as presented in the Design section have been satisfied as shown in the evaluation of the case study as well as the test cases provided. The remaining General functional requirements have been proven to be satisfied throughout the write up. Here is a summary of the functional and non-functional requirements:

- FR8. The translator should use a parser generator to parse an input file

FR8 has been satisfied by using the ANTLR parser generator. The full grammar used to parse the EROP language has been included in the appendixes section.

- FR9. The translator should be able to create an intermediate representation of the parsed input

The developed classes from the Rules Structure section serve for the creation of an intermediary representation of the parsed input. The classes in their entirety have been discussed earlier along with an UML diagram, showcasing the dependencies between them,

- FR10. The translator should be written in JAVA,
- FR11 - The translator should be of the form of a standalone GUI application, an executable script or both
- NFR1. The EROP input file should not be modified in the translation process.

These functional requirements have been satisfied, as proved by the user manual provided in the appendix section.

- NFR2. The translator should work with the latest version of AD

The proposed changes, as discussed in sections 5.2.1 and 5.2.2, make it possible for the translator to map to and support the latest implementation of Augmented Drools.

- NFR3. A user manual should be created to show how the translator can be used.

A user manual, explaining how the final solution works as well as any required input files and dependencies has been created and can be found in the appendixes section.

- NFR4. The system should be well structured, encapsulating different functionality.

The final architectural design, as presented in the Testing section, shows that the system is based on a several layer architecture, encapsulation different functionality and enforcing a modular design.

4.2.5 Evaluation of the Software Engineering aspects

The software methodology I set out to use at the beginning of the project was agile. I chose it because I was familiar with it from my time spent on a work placement as a development intern and I believe that an incremental approach would yield a better tested more robust solution.

After the completion of the project I can't say that the methodology I employed wasn't pure agile. Because of the enormous amounts of background research required to get myself up to speed with contract specification languages, electronic contracts and translations I couldn't start the normal agile sprints of design, implementation and testing right away. This resulted in

an initial long design phase, after which the agile sprints including refinements of the original design as well as incremental implementation and testing started.

The carried out sprints in my opinion increased my productivity, by setting a certain amount of work, (design, implementation or testing), that had to be done each week. This kept me engaged in the project work, not allowing me to spend too much time on theory.

The actual agile sprint length varied but it was generally between one and two weeks in length. One of the most helpful aspects of the process was the fact that implementation and testing were very close together, which was very useful in the developing and testing of the grammar needed for ANTLR.

In retrospect, I believe that my approach was suitable for the development of the chosen project. If I had to do it again, perhaps the biggest change I would make is the degree to which the current version of Augmented Drools is addressed in the project. More exposure to the latest version of AD would have ensured that the resulting translator is as accurate and as up to date as possible.

4.2.6 Skills learned

When I started work on developing a solution I had limited background knowledge of electronic contracts and anything that relates to them, including research done by the university on the topic. With the amount of background work that needed to be done in order to develop a translator I can say that my knowledge of the field has improved substantially.

When researching into translation techniques I learned a great deal about compilers, how they work and the methods they employ in order to get a translation. During that time I also learned about translation specific tools such as parser generator, syntax trees and automation tools. This opened the door to researching parser generators and ANTLR in particular. When researching how ANTLR works, I learned about different types of grammars and algorithms employed by parser generators. All that combined with the research skills and information finding process contributed greatly to the development of my ability to find relevant information and apply it in a specific way in order to accomplish a desired outcome.

Throughout the project I used JAVA as a programming language, solidifying everything I've learned over the past three years. I also did research in the development of a structured XML and how to properly parse it in an application as well as the different options available to do the parsing such as DOM and SAX parsing. In the early stages of development I researched the development of plugins for IDEs such as Eclipse and IntelliJ IDEA. The development of the final

application also helped me gain an understanding in creating executable and deployable JAVA applications in the form of scrip and a GUI.

The project as a whole gave me an experience of what it is like to carry out all the stages required in the development of small size software development project. The employed software methodology improved my time management skills and my ability to estimate how long it would require me to complete a task. Overall the final year project has been a positive experience yielding information that goes beyond of what can be covered in a taught module.

4.2.7 Conclusion

The set of aims and objectives set initially - to develop a translation tool, research various different translation techniques and their application in the development of an EROP to Augmented Drools translator, as well as the use of a specific structure, development approach and definition of test cases that showcase how the translator works with different language constructs were quite ambitious given the timeframe.

As a starting point, an extensive background research was needed given my unfamiliarity with the concept of electronic contracts and the research conducted by the university on various different topics relating to electronic contracts, contract specification techniques and compliance monitoring solutions. Next, the background research focused on translation techniques and approaches as well as different projects doing translations from different languages. During that research fundamental aspects of the translation process were discovered such as parsing, mapping and the typical architecture of compilers and language applications.

The background research served as a starting point of the architectural design of the developed solution. With a collection of techniques inspired from compilers and different research papers focusing on translations from/to rules languages, the initial design was created. Using incremental development and testing, the right grammar for the used parser generator – ANTLR was developed. The grammar evolved over the myriad test cases as well as the presented case study to reveal some of the unneeded language constructs and the need of new such so that a translation to the latest version of Augmented Drools is possible.

Given that EROP was a conceptual language with no concrete implementation, it seemed that the need to keep developing it was not a priority. This was evident when the inconsistencies

between EROP, which was unchanged since its initial introduction and the latest developed version of Augmented Drools, were discovered. One of the main challenges in the development of the translation was the fact that the latest version of the Augmented Drools implementation was not available for reference. This resulted in me having to use papers and implementation of Augmented Drools that were outdated and led to the development of mappings for obsolete methods. Fortunately the regular meetings arranged from my supervisor with developers of Augmented Drools led to the resolution of such erroneous mappings and sparked discussion about EROP.

4.2.8 Future Work

Looking back I believe that the project has been a success - I was able to complete the aims and objectives I set out to accomplish at the beginning of the project and I gained invaluable insight in academic research, software methodology and development. Even though the developed solution is able to generate translations, there are improvements that can be made to enhance its capabilities. Some of these include:

- Adding more descriptive error handling mechanism – As it currently is whenever the translator tries to parse a file, it expects the input to follow a certain format as specified in the grammar. If it doesn't find what it expects, error messages are presented that show the line of the file and character position at which the parser found an unexpected input. The error messages can be enhanced and made more descriptive and user friendly.
- Think about integration with the CCC [1][2][3][28] – Currently the translator is a stand-alone entity, separate from the CCC. It is worth exploring the cost of integrating it with the CCC and the amount of work required to do so. If it is ever integrated that would contribute to the completeness of the CCC as a contract compliance monitoring system.
- Enhancing the translator so that it can translate EROP to other Rules Languages – The developed solution was targeted at EROP to AD translation but with the used technologies in the face of ANTLR – a grammar has the capability to serve for multi-language translations. In addition to rule languages such as AD, we could explore translation to blockchain based languages such as Ethereum's Solidity. In order for that to be accomplished, the intermediate representation has to be extended and one that is closer to the new target languages has to be added. A possible alternative is ePromela[28]. Smart contracts built on blockchain and hybrid based architectures are discussed and analysed in [29][30][31].
- Integrating the translator with automated contract verification capabilities developed at Newcastle University[23][24][25][26][27].

5 References

- [1] Solaiman E, Sfyarakis I, Molina-Jimenez C. Dynamic Testing and Deployment of a Contract Monitoring Service. *In: 5th International Conference on Cloud Computing and Services Science (CLOSER 2015)*. 2015, Lisbon, Portugal: SCITEPRESS.
- [2] Strano M, Molina-Jimenez C, Shrivastava S. Implementing a Rule-Based Contract Compliance Checker. *In: Software Services for e-Business and e-Society: 9th IFIP WG 6.1 Conference on e-Business, e-Services and e-Society (I3E)*. 2009, Nancy, France: Springer.
- [3] Strano, Massimo 2008, 'Contract specification for compliance checking of business interactions', PhD Thesis, Newcastle University, Newcastle upon Tyne.
- [4] B Nicholas, *An Introduction to Roman Law* (Clarendon 1963) 165-193.
- [5] Yao-Hua, Tan. 'Doclog: An Electronic Contract Representation Language'. *Database and Expert Systems Applications, 2000. Proceedings. 11th International Workshop (2000)*: 1069 – 1073.
- [6] Techopedia.com,. 'What Is A Transaction Process System (TPS) - Definition From Techopedia'. Web. May 2015.
- [7] EDI Translator. [http:// Altova.com](http://Altova.com). May 2015.
- [8] The CONTRACT Project. <http://www.ist-contract.org>.
- [9] S. Miles, N. Oren, M. Luck, S. Modgil, N. Faci, C. Holt, and G. Vickers. Modelling and Administration of Contract-based Systems. *In Proceedings of the AISB 2008 Symposium on Behaviour Regulation in Multi-agent Systems*, pages 19–24, 2008.
- [10] N. Faci, S. Modgil, N. Oren, F. Meneguzzi, S. Miles, and M. Luck. Towards a Monitoring Framework for Agent-based Contract Systems. *In Proceedings of the 12th international workshop on Cooperative Information Agents XII*, pages 292–305. Springer, 2008.
- [11] S. Panagiotidi, J. Vazquez-Salceda, S. Alvarez-Napagao, S. Ortega-Martorell, S. Willmott, R. Confalonieri, and P. Storms. Intelligent Contracting Agents Language. *Behaviour Regulation in MAS, AISB*, pages 49–55, 2008.
- [12] RuleML. The RuleML Markup Initiative. <http://www.ruleml.org>, Apr 2005.
- [13] M.B. Juric. *Business Process Execution Language for Web Services BPEL and BPEL4WS 2nd Edition*. Packt Publishing, 2006.
- [14] P. Gama and P. Ferreira. Obligation Policies: an Enforcement Platform. *In Sixth IEEE International Workshop on Policies for Distributed Systems and Networks*, 2005, pages 203–212, 2005.

- [15] C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes. SPL: An Access Control Language for Security Policies with Complex Constraints. In *Proceedings of the Network and Distributed System Security Symposium*, pages 89–107, 2001.
- [16] Miles, S., Oren, N., Luck, M., Modgil, S., Faci, N., Holt, C., & Vickers, G. (2008). *Modelling and Administration of Contract-Based Systems*. In *Proceedings of the AISB 2008 Symposium on Behaviour Regulation in Multi-Agent Systems*. (pp. 19 - 24). Unknown Publisher.
- [17] G., Aiello. 'Inferring Business Rules From Natural Language Expressions'. *Service-Oriented Computing and Applications (SOCA)*, 2014 IEEE 7th International Conference (2014): 131 - 136.
- [18] MacLennan, Bruce J. (1987). *Principles of Programming Languages*. Oxford University Press. p. 1.
- [19] Aho, Alfred V. (2013) *Compilers; Principles, Techniques And Tools*, By Alfred V. Addison Wesley.
- [20] Grune, Dick (1999). *Parsing Techniques: A Practical Guide*. US: Springer.
- [21] Knuth, Donald E. 'On The Translation Of Languages From Left To Right'. *Information and Control* 8.6 (1965): 607-639.
- [22] Parr, Terence. *The Definitive ANTLR 4 Reference*.
- [23] Solaiman E, Sun W, Molina-Jimenez C. A Tool for the Automatic Verification of BPMN Choreographies. *In: 12th IEEE International Conference on Services Computing (SCC)*. 2015, New York City, NY, USA: IEEE.
- [24] Solaiman E, Molina-Jimenez C, Shrivastava S. Model checking correctness properties of electronic contracts. *In: Service-Oriented Computing - ICSOC 2003*. 2003, Trento, Italy: Springer.
- [25] Molina-Jimenez C, Shrivastava S, Solaiman E, Warne J. Contract Representation for Run-time Monitoring and Enforcement. *In: IEEE International Conference on E-Commerce*. 2003, Newport Beach, California: IEEE.
- [26] Molina-Jimenez C, Shrivastava SK, Solaiman EM, Warne JP. Run-time monitoring and enforcement of electronic contracts. *Electronic Commerce Research and Applications* 2004, 3(2), 108-125.
- [27] Solaiman E, Sfyarakis I, Molina-Jimenez C. High Level Model Checker Based Testing Of Electronic Contracts. *In: Cloud Computing and Services Science*. Springer-Verlag, 2016, pp.193-215.
- [28] Solaiman E, Sfyarakis I, Molina-Jimenez C. A State Aware Model and Architecture for the Monitoring and Enforcement of Electronic Contracts. *In: 18th IEEE Conference on Business Informatics (CBI)*. 2016, Paris, France: IEEE.
- [29] Molina-Jimenez C, Solaiman E, Sfyarakis I, Ng I, Crowcroft J. On and Off-Blockchain Enforcement Of Smart Contracts. *EUROPAR 2018 24th International European Conference on Parallel and Distributed*. Turin, Italy: Springer.
- [30] Molina-Jimenez C, Sfyarakis I, Solaiman E, Ng I, Wong M, Chun A, Crowcroft J. Implementation of Smart Contracts Using Hybrid Architectures with On and Off-Blockchain Components. *In: The 8th IEEE International Symposium on Cloud and Services Computing (IEEE SC2) 2018*. 2018, Paris, France: IEEE.
- [31] Solaiman E, Wike T, Sfyarakis I. Implementation And Evaluation Of Smart Contracts Using A Hybrid On And Off-Blockchain Architecture. *Concurrency and Computation Practice and Experience* 2020.

6 Appendixes

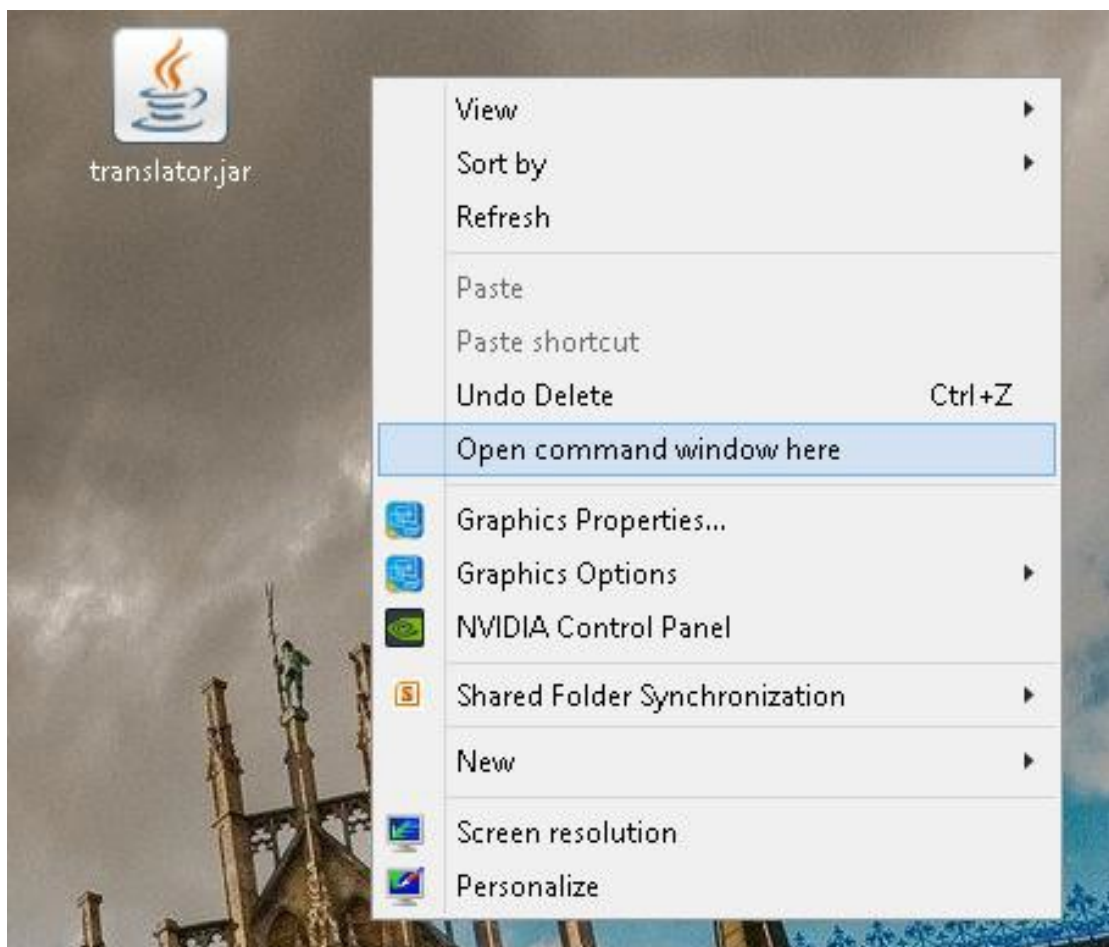
6.1 User Manual

The produced piece of software was written in JAVA [conforming to FR10 and FR11] and is the form of an executable jar file with no GUI. It contains all the external libraries and files needed so that it can be run as a stand-alone file. The only dependency on the machine that it will be run is that it has JRE installed (preferably the latest version).

Running the JAR:

The jar script can be executed through the command line of the used operation system. Here is an example using Windows 8.1

Open the command line windows at the place where the Jar file is located by holding down the shift key on the keyboard and right clicking with the mouse. From the menu select “Open command window here”.

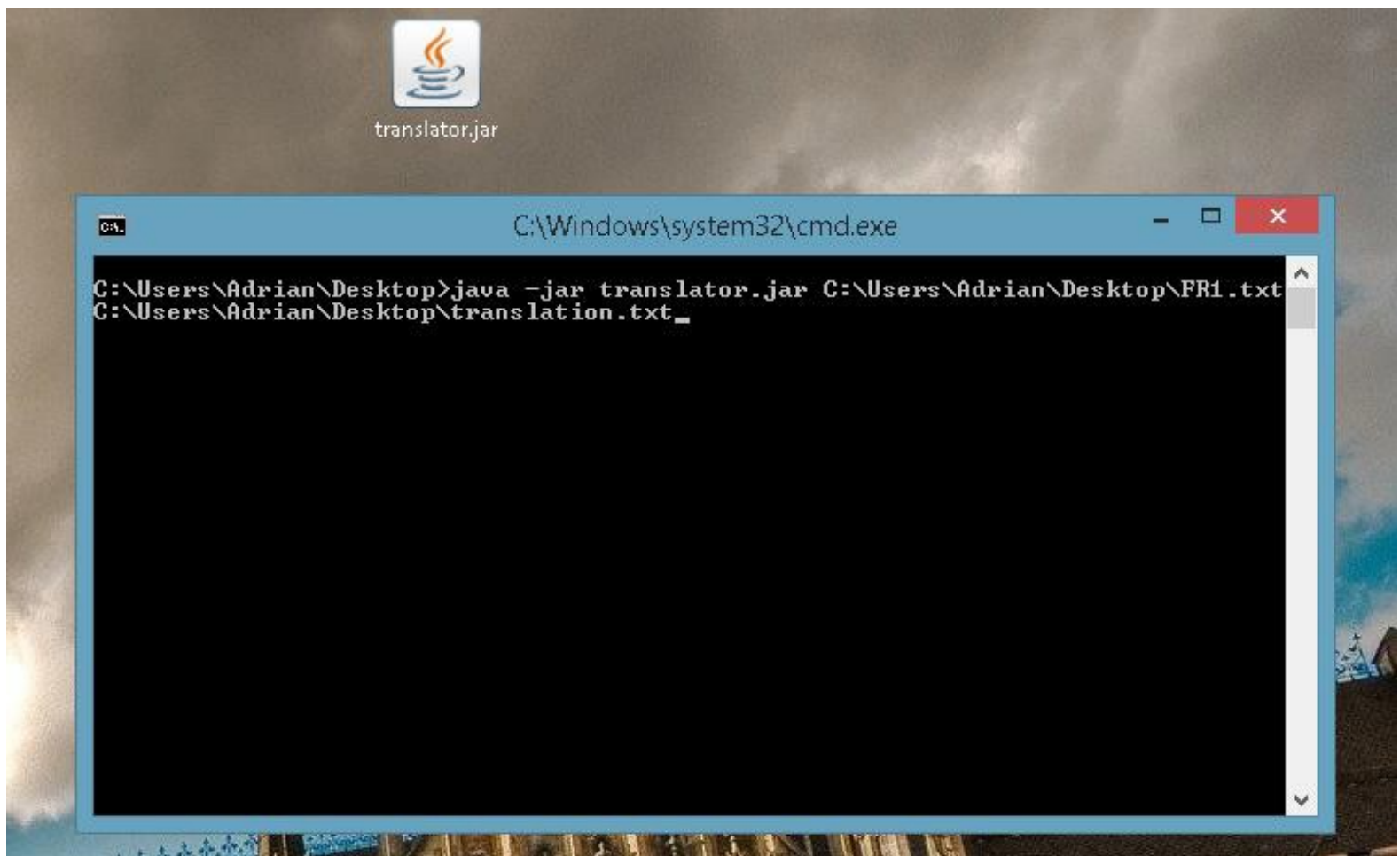


Once the command line window is opened the script can be executed by specifying the file name and using the standard command jar, provided by the JRE. In order to trigger a translation two parameters have to be provided – the first one representing the path to the input file in EROP and the second file representing the path to the output file that will contain the translation to Augmented Drools

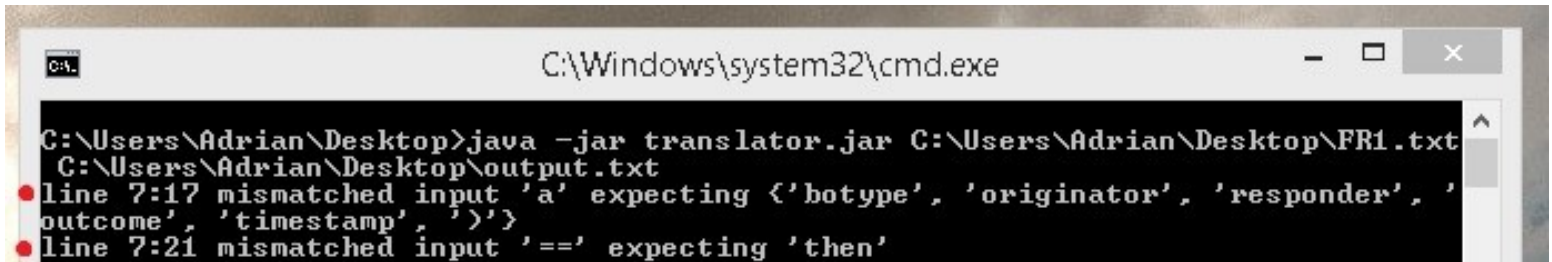
The command to run the application is:

```
Java -jar translator.jar PathOfEropFile PathOfOutputFile
```

Where *PathOfEropFile* is the file path on the system to the input file and *PathOfOutputFile* is the file path to the output file.



If an exception occurs it will be noted in the console window. If the EROP file specified contains syntactical errors they will also be noted in the console window. The following picture

A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The prompt shows the execution of a Java program: `C:\Users\Adrian\Desktop>java -jar translator.jar C:\Users\Adrian\Desktop\FR1.txt C:\Users\Adrian\Desktop\output.txt`. The output of the program is displayed on the next two lines, with red dots at the beginning of each line: `line 7:17 mismatched input 'a' expecting {'botype', 'originator', 'responder', 'outcome', 'timestamp', '}'}` and `line 7:21 mismatched input '==' expecting 'then'`.

```
C:\Windows\system32\cmd.exe
C:\Users\Adrian\Desktop>java -jar translator.jar C:\Users\Adrian\Desktop\FR1.txt
C:\Users\Adrian\Desktop\output.txt
line 7:17 mismatched input 'a' expecting {'botype', 'originator', 'responder', '
outcome', 'timestamp', '}'
line 7:21 mismatched input '==' expecting 'then'
```

represents the feedback that is given in the case of syntactically incorrect input file.

It indicates that the problem with the input file occurs on the seventh line and it states that it found 'a' but according to the specified grammar it should have found any of botype/originator/responder/outcome/timestamp.

If there aren't errors of any type a 'Writing to file finished' message will be printed that would indicate that the translation has been completed.

6.2 Mapping FR tests

- FR1. The translator should always include the classes from the EROP Ontology.

Input:

```
roleplayer buyer,seller;  
businessoperation BuyRequest,Payment,BuyConfirm,BuyReject,Cancellation;
```

Output:

```
package BuyerStoreContractEx  
import uk.ac.ncl.erop.*;  
import uk.ac.ncl.logging.CCCLogger;  
  
global RelevanceEngine engine;  
global EventLogger logger;  
global TimingMonitor timingMonitor;  
  
global RolePlayer buyer;  
global ROPSet ropBuyer  
global RolePlayer seller;  
global ROPSetropSeller  
global BusinessOperation buyRequest;  
global BusinessOperation payment;  
global BusinessOperation buyConfirm;  
global BusinessOperation buyReject;  
global BusinessOperation cancelation;
```

- FR2. The translator should create any instances of the ontology classes used in the rule referencing.

Input:

roleplayer buyer,seller;
businessoperation BuyRequest,Payment,BuyConfirm,BuyReject,Cancellation;

Output:

```
package BuyerStoreContractEx  
import uk.ac.ncl.erop.*;  
import uk.ac.ncl.logging.CCCLogger;
```

```
global RelevanceEngine engine;  
global EventLogger logger;  
global TimingMonitor timingMonitor;
```

```
global RolePlayer buyer;  
global ROPSet ropBuyer  
global RolePlayer seller;  
global ROPSet ropSeller  
global BusinessOperation buyRequest;  
global BusinessOperation payment;  
global BusinessOperation buyConfirm;  
global BusinessOperation buyReject;  
global BusinessOperation cancelation;
```

- FR3. The translator should create ROP sets for every declared role player.

Input:

roleplayer buyer,seller;
businessoperation BuyRequest,Payment,BuyConfirm,BuyReject,Cancellation;

Output:

```
package BuyerStoreContractEx  
import uk.ac.ncl.erop.*;  
import uk.ac.ncl.logging.CCCLogger;
```

```
global RelevanceEngine engine;  
global EventLogger logger;  
global TimingMonitor timingMonitor;
```

```
global RolePlayer buyer;  
global ROPSet ropBuyer  
global RolePlayer seller;  
global ROPSet ropSeller  
global BusinessOperation buyRequest;  
global BusinessOperation payment;  
global BusinessOperation buyConfirm;  
global BusinessOperation buyReject;  
global BusinessOperation cancelation;
```

- FR4. The translator should maintain integrity of style conventions when translating names of business operations or composite obligations (capital in EROP but lowercase in AD)

Input:

```
roleplayer buyer,seller;  
businessoperation BuyRequest,Payment,BuyConfirm,BuyReject,Cancelation;
```

Output:

```
package BuyerStoreContractEx  
import uk.ac.ncl.erop.*;  
import uk.ac.ncl.logging.CCCLogger;
```

```
global RelevanceEngine engine;  
global EventLogger logger;  
global TimingMonitor timingMonitor;
```

```
global RolePlayer buyer;  
global ROPSet ropBuyer  
global RolePlayer seller;  
global ROPSet ropSeller  
global BusinessOperation buyRequest;  
global BusinessOperation payment;  
global BusinessOperation buyConfirm;  
global BusinessOperation buyReject;  
global BusinessOperation cancelation;
```

- FR5. The translator should first translate the declaration section and only then the rules section.

Input:

```
roleplayer buyer,seller;
businessoperation BuyRequest,Payment,BuyConfirm,BuyReject,Cancelation;
compoblig ReactToBuyRequest(BuyConfirm,BuyReject);
```

```
rule "BuyRequestReceived"
  when e matches (botype == BUYREQ,originator == buyer,responder == store,outcome ==
  success)
    BuyRequest in buyer.rights
  then
    buyer.rights -= BuyRequest(seller)
    seller.obligs += ReactToBuyRequest(buyer)
  end
```

```
rule "BuyRequestBnessFailures"
  when e matches (botype == BUYREQ,originator == buyer,responder == store,outcome == tecFail)
    BuyRequest in buyer.rights
  then
    if (BuyRequest.BizFail == false )
      then BuyRequest.BizFail == true
      else reset buyer reset seller
    endif
  end
```

Output:

```
package BuyerStoreContractEx
import uk.ac.ncl.erop.*;
import uk.ac.ncl.logging.CCCLogger;
```

```
global RelevanceEngine engine;
global EventLogger logger;
global TimingMonitor timingMonitor;
```

```
global RolePlayer buyer;
global ROPSetropBuyer
global RolePlayer seller;
global ROPSetropSeller
global BusinessOperation buyRequest;
global BusinessOperation payment;
```

```
global BusinessOperation buyConfirm;
global BusinessOperation buyReject;
global BusinessOperation cancelation;
```

```
rule "BuyRequestReceived"
  when $e: Event(type == "buyreq",originator == "buyer",responder == "store",status ==
    "success")
    eval(ropBuyer.matchesRights(buyRequest))
  then
    ropBuyer.removeRight(buyRequest, seller)
    BusinessOperation[] bos= {buyConfirm, buyReject};
    ropSeller.addObligation(reactToBuyRequest, bos,buyer)
end
```

```
rule "BuyRequestBnessFailuresIfThen"
  when $e: Event(type == "buyreq",originator == "buyer",responder == "store",status ==
    "tecfail")
    eval(buyRequest.getBusinessFailure()== false)
    eval(ropBuyer.matchesRights(buyRequest))
  then
    buyRequest.setBusinessFailure (true)
end
```

```
rule "BuyRequestBnessFailuresIfElse"
  when $e: Event(type == "buyreq",originator == "buyer",responder == "store",status ==
    "tecfail")
    eval((buyRequest.getBusinessFailure()== false)== false)
    eval(ropBuyer.matchesRights(buyRequest))
  then
    ropBuyer.reset()
    ropSeller.reset()
end
```

- FR6. The translator should correctly translate keywords in EROP to the corresponding method calls in AD

Input:

```
roleplayer buyer,seller;
businessoperation BuyRequest,Payment,BuyConfirm,BuyReject,Cancellation;
compoblig ReactToBuyRequest(BuyConfirm,BuyReject);
```

```
rule "BuyRequestReceived"
  when e matches (botype == BUYREQ,originator == buyer,responder == store,outcome ==
  success)
    BuyRequest in buyer.rights
  then
    buyer.rights -= BuyRequest(seller)
    seller.obligs += ReactToBuyRequest(buyer)
  end
```

```
rule "BuyRequestBnessFailures"
  when e matches (botype == BUYREQ,originator == buyer,responder == store,outcome == tecFail)
    BuyRequest in buyer.rights
  then
    if (BuyRequest.BizFail == false )
      then BuyRequest.BizFail == true
      else reset buyer reset seller
    endif
  end
```

Output:

```
package BuyerStoreContractEx
import uk.ac.ncl.erop.*;
import uk.ac.ncl.logging.CCCLogger;

global RelevanceEngine engine;
global EventLogger logger;
global TimingMonitor timingMonitor;

global RolePlayer buyer;
global ROPSet ropBuyer
global RolePlayer seller;
global ROPSetropSeller
global BusinessOperation buyRequest;
```



```
global BusinessOperation payment;
global BusinessOperation buyConfirm;
global BusinessOperation buyReject;
global BusinessOperation cancelation;
```

```
rule "BuyRequestReceived"
```

```
  when $e: Event(type == "buyreq",originator == "buyer",responder == "store",status ==
    "success")
```

```
    eval(ropBuyer.matchesRights(buyRequest))
```

```
  then
```

```
    ropBuyer.removeRight(buyRequest, seller)
```

```
    BusinessOperation[] bos= {buyConfirm, buyReject};
```

```
    ropSeller.addObligation(reactToBuyRequest, bos,buyer)
```

```
end
```

```
rule "BuyRequestBnessFailuresIfThen"
```

```
  when $e: Event(type == "buyreq",originator == "buyer",responder == "store",status ==
    "tecfail")
```

```
    eval(buyRequest.getBusinessFailure() == false)
```

```
    eval(ropBuyer.matchesRights(buyRequest))
```

```
  then
```

```
    buyRequest.setBusinessFailure (true)
```

```
end
```

```
rule "BuyRequestBnessFailuresIfElse"
```

```
  when $e: Event(type == "buyreq",originator == "buyer",responder == "store",status ==
    "tecfail")
```

```
    eval((buyRequest.getBusinessFailure() == false) == false)
```

```
    eval(ropBuyer.matchesRights(buyRequest))
```

```
  then
```

```
    ropBuyer.reset()
```

```
    ropSeller.reset()
```

```
end
```

- FR7. The translator should split rules in EROP that have an f-then-else section into two rules in AD.

Input:

```
roleplayer buyer,seller;
businessoperation BuyRequest,Payment,BuyConfirm,BuyReject,Cancelation;
compoblig ReactToBuyRequest(BuyConfirm,BuyReject);
```

```
rule "BuyRequestReceived"
  when e matches (botype == BUYREQ,originator == buyer,responder == store,outcome ==
  success)
    BuyRequest in buyer.rights
  then
    buyer.rights -= BuyRequest(seller)
    seller.obligs += ReactToBuyRequest(buyer)
  end
```

```
rule "BuyRequestBnessFailures"
  when e matches (botype == BUYREQ,originator == buyer,responder == store,outcome == tecFail)
    BuyRequest in buyer.rights
  then
    if (BuyRequest.BizFail == false )
      then BuyRequest.BizFail == true
      else reset buyer reset seller
    endif
  end
```

Output:

```
package BuyerStoreContractEx
import uk.ac.ncl.erop.*;
import uk.ac.ncl.logging.CCCLogger;

global RelevanceEngine engine;
global EventLogger logger;
global TimingMonitor timingMonitor;

global RolePlayer buyer;
global ROPSet ropBuyer
global RolePlayer seller;
global ROPSet ropSeller
global BusinessOperation buyRequest;
```

```
global BusinessOperation payment;
global BusinessOperation buyConfirm;
global BusinessOperation buyReject;
global BusinessOperation cancelation;
```

```
rule "BuyRequestReceived"
  when $e: Event(type == "buyreq",originator == "buyer",responder == "store",status ==
    "success")
    eval(ropBuyer.matchesRights(buyRequest))
  then
    ropBuyer.removeRight(buyRequest, seller)
    BusinessOperation[] bos= {buyConfirm, buyReject};
    ropSeller.addObligation(reactToBuyRequest, bos,buyer)
end
```

```
rule "BuyRequestBnessFailuresIfThen"
  when $e: Event(type == "buyreq",originator == "buyer",responder == "store",status ==
    "tecfail")
    eval(buyRequest.getBusinessFailure() == false)
    eval(ropBuyer.matchesRights(buyRequest))
  then
    buyRequest.setBusinessFailure (true)
end
```

```
rule "BuyRequestBnessFailuresIfElse"
  when $e: Event(type == "buyreq",originator == "buyer",responder == "store",status ==
    "tecfail")
    eval((buyRequest.getBusinessFailure() == false) == false)
    eval(ropBuyer.matchesRights(buyRequest))
  then
    ropBuyer.reset()
    ropSeller.reset()
end
```

6.3 Full translation of the Contract presented in section 4

```
package BuyerStoreContractEx
import uk.ac.ncl.erop.*;
import uk.ac.ncl.logging.CCCLogger;

global RolePlayer buyer;
global ROPSet ropBuyer
global RolePlayer seller;
global ROPSet ropSeller
global BusinessOperation buyRequest;
global BusinessOperation payment;
global BusinessOperation buyConfirm;
global BusinessOperation buyReject;
global BusinessOperation cancelation;

rule "BuyRequestReceived"
  when $e: Event(type == "buyreq",originator == "buyer",responder ==
    "store",status == "success")
    eval(ropBuyer.matchesRights(buyRequest))
  then
    ropBuyer.removeRight(buyRequest, seller)
    BusinessOperation[] bos = {buyConfirm, buyReject};
    ropSeller.addObligation(reactToBuyRequest, bos,buyer)
end

rule "BuyRequestBnessFailuresIfThen"
  when $e: Event(type == "buyreq",originator == "buyer",responder ==
"store",status == "tecfail")
    eval(buyRequest.getBusinessFailure() == false)
    eval(ropBuyer.matchesRights(buyRequest))
  then
    buyRequest.setBusinessFailure (true)
end

rule "BuyRequestBnessFailuresIfElse"
  when $e: Event(type == "buyreq",originator == "buyer",responder ==
"store",status == "tecfail")
    eval((buyRequest.getBusinessFailure() == false) == false)
    eval(ropBuyer.matchesRights(buyRequest))
  then
    ropBuyer.reset()
    ropSeller.reset()
end
```

```

rule "BuyRequestRejected"
    when $e: Event(type == "buyrej",originator == "store",responder ==
"buyer",status == "success")
        eval(ropSeller.matchesObligs(reactToBuyRequest))
    then
        ropSeller.removeObligation(reactToBuyRequest, buyer)
end

rule "BuyRequestRejectedFailuresIfThen"
    when $e: Event(type == "buyrej",originator == "store",responder ==
"buyer",status == "tecfail")
        eval(buyConfirm.getBusinessFailure() == false)
        eval(ropSeller.matchesObligs(reactToBuyRequest))
    then
        buyConfirm.setBusinessFailure (true)
end

rule "BuyRequestRejectedFailuresIfElse"
    when $e: Event(type == "buyrej",originator == "store",responder ==
"buyer",status == "tecfail")
        eval((buyConfirm.getBusinessFailure() == false) == false)
        eval(ropSeller.matchesObligs(reactToBuyRequest))
    then
        ropBuyer.reset()
        ropSeller.reset()
end

rule "BuyRequestConfirmation"
    when $e: Event(type == "buyconf",originator == "seller",responder ==
"buyer",status == "success")
        eval(ropBuyer.matchesObligs(reactToBuyRequest))
    then
        ropSeller.removeObligation(reactToBuyRequest, buyer)
        ropBuyer.addObligation(payment, seller)
        ropBuyer.addRight(cancellation, seller)
end

```

```

rule "BuyRequestConfirmationFailuressIfThen"
  when $e: Event (type == "buyconf",originator == "seller",responder ==
    "buyer",status == "tecfail")
    eval (buyConfirm.getBusinessFailure() == false)
    eval (ropSeller.matchesObligs (reactToBuyRequest))
  then
    buyConfirm.setBusinessFailure (true)
end

```

```

rule "BuyRequestConfirmationFailuressIfElse"
  when $e: Event (type == "buyconf",originator == "seller",responder ==
    "buyer",status == "tecfail")
    eval ((buyConfirm.getBusinessFailure() == false) == false)
    eval (ropSeller.matchesObligs (reactToBuyRequest))
  then
    ropBuyer.reset ()
    ropSeller.reset ()
end

```

```

rule "PaymentReceived"
  when $e: Event (type == "buypay",originator == "buyer",responder ==
    "store",status == "success")
    eval (ropBuyer.matchesObligs (payment))
  then
    ropBuyer.removeObligation (payment, seller)
    ropBuyer.removeRight (cancellation, seller)
end

```

```

rule "PaymentReceivedBFailuresIfThen"
  when $e: Event (type == "buypay",originator == "buyer",responder ==
    "store",status == "tecfail")
    eval (payment.getBusinessFailure() == false)
    eval (ropBuyer.matchesObligs (payment))
  then
    payment.setBusinessFailure (true)
end

```

```

rule "PaymentReceivedBFailuresIfElse"
  when $e: Event (type == "buypay", originator == "buyer", responder ==
    "store", status == "tecfail")
    eval ((payment.getBusinessFailure() == false) == false)
    eval (ropBuyer.matchesObligs (payment))
  then
    ropBuyer.reset ()
    ropSeller.reset ()
end

rule "BuyCancellation"
  when $e: Event (type == "buycanc", originator == "buyer", responder ==
    "store", status == "success")
    eval (ropBuyer.matchesRights (cancellation))
  then
    ropBuyer.removeRight (cancellation, seller)
    ropBuyer.removeObligation (payment, seller)
end

rule "CancellationBFailuresIfThen"
  when $e: Event (type == "buycanc", originator == "buyer", responder ==
    "store", status == "tecfail")
    eval (cancellation.getBusinessFailure() == false)
    eval (ropBuyer.matchesRights (cancellation))
  then
    cancellation.setBusinessFailure (true)
end

rule "CancellationBFailuresIfElse"
  when $e: Event (type == "buycanc", originator == "buyer", responder ==
    "store", status == "tecfail")
    eval ((cancellation.getBusinessFailure() == false) == false)
    eval (ropBuyer.matchesRights (cancellation))
  then
    ropBuyer.reset ()
    ropSeller.reset ()
end

```

6.4 Original Contract from section 4 in Augmented Drools

```
package BuyerStoreContractEx

// Import Java classes for EROP support
import uk.ac.ncl.erop.*;
import uk.ac.ncl.logging.CCCLogger;
// Global variables (persistent objects passed from outside)
global RelevanceEngine engine;
global EventLogger logger;
global TimingMonitor timingMonitor;

global RolePlayer buyer;
global RolePlayer seller;
global ROPSet ropBuyer;
global ROPSet ropSeller;

global BusinessOperation buyRequest;
global BusinessOperation payment;
global BusinessOperation buyConfirm;
global BusinessOperation buyReject;
global BusinessOperation cancelation;

global Responder responder;

global CCCLogger ccclogger;

/* Rule 0: initialize the ROP sets for buyer and seller.
 * This rule is launched only when the contract is set up.
 * the buyer
 * starts with the right to submit a buy request. */

rule "Initialization"
    when
        $e: Event (type == "init")
    then
        ropBuyer.addRight(buyRequest, seller, (String)null);
    end

/* Rule 1: having received a Buy Request event from the buyer, his right to
submit another
 * is temporarily revoked until the current one is completed. The seller gains
 * an obligation to either accept or reject the Buy Request. */

rule "Buy Request Received"
```



```

    when
//      Verify type of event, originator, and responder
    $e: Event(type=="BUYREQ", originator=="buyer",
responder=="store", status=="success")
        eval(ropBuyer.matchesRights(buyRequest))
    then

//      Remove buyer's right to place other Buy Requests
ropBuyer.removeRight(buyRequest, seller);

//      Add seller's obligation to either accept or reject order
BusinessOperation[] bos = {buyConfirm, buyReject};
ropSeller.addObligation("React To Buy Request", bos, buyer,
60,2);

end

rule "Buy Request Business 1st Failure"
    when
        // Verify type of event, originator, and responder
        $e: Event(type=="BUYREQ", originator=="buyer",
responder=="store", status=="tecfail")
        eval(ropBuyer.matchesRights(buyRequest) &&
buyRequest.getBusinessFailure()== false )

    then
        buyRequest.setBusinessFailure(true);
    end

/*
rule "Buy Request Business 2nd Failure"
    when
        // Verify type of event, originator, and responder
        $e: Event(type=="BUYREQ", originator=="buyer",
responder=="store", status=="tecfail")
        eval(ropBuyer.matchesRights(buyRequest) &&
buyRequest.getBusinessFailure()== true)

    then
        ropBuyer.reset();
        ropSeller.reset();
    end
*/
/* Rule 2: having received a reject Buy Request event from the seller, the
pending obligation
* is satisfied. Restore buyer's right to submit Buy Requests.
*/

rule "Buy Request Rejected"
    when

```

```

        $e: Event(type=="BUYREJ", originator=="store",
responder=="buyer", status=="success")
        eval(ropSeller.matchesObligations("React To Buy Request"));
    then
        // Buyer's Obligation is satisfied, remove it
        ropSeller.removeObligation("React To Buy Request", buyer);

        // Restore buyer's right to submit other Buy Requests
        //ropBuyer.addRight(buyRequest, seller, (String)null);
end

rule "Buy Request Rejected Business 1st Failure"
    when
        $e: Event(type=="BUYREJ", originator=="store",
responder=="buyer", status=="tecfail")
        eval(ropSeller.matchesObligations("React To Buy Request") &&
buyReject.getBusinessFailure()==false)
    then
        buyReject.setBusinessFailure(true);
end

/*
rule "Buy Request Rejected Business 2nd Failure"
    when
        $e: Event(type=="BUYREJ", originator=="store",
responder=="buyer", status=="tecfail")
        eval(ropSeller.matchesObligations("React To Buy Request") &&
buyReject.getBusinessFailure()==true)
    then

        ropBuyer.reset();
        ropSeller.reset();

end
*/
/* Rule 3: having received an accept Buy Request event from the seller, the
pending obligation
* is satisfied. New obligation on buyer to pay seller. */
rule "Buy Request Confirmation"
    when
        $e: Event(type=="BUYCONF", originator=="store",
responder=="buyer", status=="success")
        eval(ropSeller.matchesObligations("React To Buy Request"));
    then
        // Buyer's Obligation is satisfied, remove it
        ropSeller.removeObligation("React To Buy Request", buyer);

        ropBuyer.addObligation(payment, seller);
        ropBuyer.addRight(cancelation, seller);
end

```

```

rule "Buy Request Confirmation 1st Business Failure"
    when
        $e: Event(type=="BUYCONF", originator=="store",
responder=="buyer", status=="tecfail")
        eval(ropSeller.matchesObligations("React To Buy Request") &&
buyConfirm.getBusinessFailure()==false)
    then
        buyConfirm.setBusinessFailure(true);
end
/*
rule "Buy Request Confirmation 2nd Business Failure"
    when
        $e: Event(type=="BUYCONF", originator=="store",
responder=="buyer", status=="tecfail")
        eval(ropSeller.matchesObligations("React To Buy Request") &&
buyConfirm.getBusinessFailure()==true)
    then
        ropBuyer.reset();
        ropSeller.reset();
end
*/
// Rule 5: buyer pays. Obligation satisfied, The buyer regains the right to
submit Buy Requests.
rule "Payment Received"
    when
        $e: Event(type=="BUYPAY", originator=="buyer",
responder=="store", status=="success")
        eval(ropBuyer.matchesObligations(payment))
    then
        // Buyer's Obligation is satisfied, remove it.
        ropBuyer.removeObligation(payment, seller);
        ropBuyer.removeRight(cancelation, seller);
end

rule "Payment 1st Business Failure"
    when
        $e: Event(type=="BUYPAY", originator=="buyer",
responder=="store", status=="tecfail")
        eval(ropBuyer.matchesObligations(payment) &&
payment.getBusinessFailure()==false)
    then
        payment.setBusinessFailure(true);
end
/*
rule "Payment 2nd Business Failure"
    when
        $e: Event(type=="BUYPAY", originator=="buyer",
responder=="store", status=="tecfail")

```

```

        eval (ropBuyer.matchesObligations (payment)  &&
payment.getBusinessFailure ()==true)
        then

            ropBuyer.reset ();
            ropSeller.reset ();

end
*/

rule "Buy cancelation"
    when
        $e: Event (type=="BUYCANC", originator=="buyer",
responder=="store", status=="success")
        eval (ropBuyer.matchesRights (cancelation) )
    then
        // Buyer's Obligation is satisfied, remove it.
        ropBuyer.removeRight (cancelation, seller);
        ropBuyer.removeObligation (payment, seller);
    end

end

rule "Cancelation 1st Business Failure"
    when
        $e: Event (type=="BUYCANC", originator=="buyer",
responder=="store", status=="tecfail")
        eval (ropBuyer.matchesRights (cancelation)  &&
cancelation.getBusinessFailure ()== false)
    then
        // Buyer's Obligation is satisfied, remove it.
        cancelation.setBusinessFailure (true);
    end

end
/*
rule "Cancelation 2nd Business Failure"
    when
        $e: Event (type=="BUYCANC", originator=="buyer",
responder=="store", status=="tecfail")
        eval (ropBuyer.matchesRights (cancelation)  &&
cancelation.getBusinessFailure ()== true)
    then
        // Buyer's Obligation is satisfied, remove it.

        ropBuyer.reset ();
        ropSeller.reset ();

end
*/

```

6.5 Formal refined grammar of EROP

Based on the work presented in [3]

```
// Grammar for EROP language
grammar Eropep;
// Package specification
@header { package com.translator.antlr; }
// Contract definition
contractDocument: WS? declarationSection WS? ruleSet WS?;
// Structure of the declaration section
declarationSection: declaration (WS declaration)*;
declaration: businessOpDeclaration|roleplayerDeclaration|compobligDeclaration;
//businessOpDeclaration: BUSINESSOP WS bopIdentifier (COMMA WS? bopIdentifier)* SEMICOLON;
//roleplayerDeclaration: ROLEPLAYER WS roleplayeridentifyer (COMMA WS? roleplayeridentifyer)*
SEMICOLON;
compobligDeclaration: COMPOBLIG WS upalphanum BRA upalphanum (COMMA WS? upalphanum)+
KET SEMICOLON;

businessOpDeclaration
: BUSINESSOP WS upalphanum (COMMA upalphanum)* SEMICOLON;
roleplayerDeclaration
: ROLEPLAYER WS alphanum (COMMA alphanum)* SEMICOLON;

// Rule set structure
ruleSet : singlerule (WS singlerule)*;

// Rule structure
singlerule: RULE WS rulename WS WHEN WS lhs WS THEN WS rhs WS END;
// : 'rule' WS rulename WS lhs WS rhs WS 'end';

rulename: "\"" upalphanum "\"";

// Left hand side structure
lhs : eventmatch WS BRA (eventMcond COMMA?)* KET (WS constraint)*;

eventmatch: alphanum WS MATCHES;

eventMcond: field WS? oper WS? (alphanum|upalphanum);

rolePlayerConstraintIssuer: (ORIGINATOR|RESPONDER);
field: (BOTYPE|OUTCOME|ORIGINATOR|RESPONDER|TIMESTAMP);
oper: (EQUALS|NOTEQ);
timeOperators: (EQUALS|NOTEQ|BEFORE|AFTER);
```

```

rangeOperators: (IN|NOTIN);
andOR: (AND|OR);
bool: (BOOLEANTRUE|BOOLEANFALSE);

constraint: attributeConstraint | historicalQuery | ropConstraint;

attributeConstraint: roleplayerConstraint | outcomeConstraint | timeConstraint;
roleplayerConstraint: alphanum DOT rolePlayerConstraintIssuer WS? oper WS? alphanum;
outcomeConstraint: upalphanum DOT outcome WS? oper WS? bool;
timeConstraint: timeDirectComparison | timePartialComparison;

timeDirectComparison: alphanum DOT TIMESTAMP WS? timeOperators WS? absoluteTime;

timePartialComparison: alphanum DOT dayKey WS? oper WS? dayOfWeek
|alphanum DOT dayKey WS? rangeOperators WS? dayRange
|alphanum DOT dateKey WS? timeOperators WS? dateIdent
|alphanum DOT dateKey WS? rangeOperators WS? dateRange
|alphanum DOT monthKey WS? timeOperators WS? monthIdent
|alphanum DOT monthKey WS? rangeOperators WS? monthRange
|alphanum DOT yearKey WS? timeOperators WS? yearIdent
|alphanum DOT yearKey WS? rangeOperators WS? yearRange;

dayKey: DAY;
dayOfWeek: WEEKDAY;
dateKey: DATE;
dateIdent: DIGIT DIGIT;
monthKey: MONTH;
monthIdent: MONTHID;
yearKey: YEAR;
yearIdent: DIGIT DIGIT DIGIT DIGIT;
dayRange: SQUAREBRA WEEKDAY DOT DOT WEEKDAY SQUAREKET;
dateRange: SQUAREBRA DIGIT DIGIT DOT DOT DIGIT DIGIT SQUAREKET;
monthRange: SQUAREBRA MONTHID DOT DOT MONTHID SQUAREKET;
yearRange: SQUAREBRA DIGIT DIGIT DIGIT DIGIT DOT DOT DIGIT DIGIT DIGIT DIGIT SQUAREKET;

historicalQueryOp: (HAPPENED|CTHAPPENED);

historicalQuery: historicalQueryOp WS? BRA upalphanum COMMA WS? alphanum
COMMA WS? alphanum COMMA WS? genericString COMMA WS? outcome KET;

ropConstraint: upalphanum WS? rangeOperators WS? alphanum DOT ropset;

//Right hand side structure
rhs : rhsaction (WS? rhsActionNoIfs)*;

rhsActionNoIfs: (termaction|passaction|resetaction|addRemAction|outcomeConstraint);

```

```

rhsaction:      (ifstatement|termaction|passaction|resetaction|addRemAction|outcomeConstraint);
ifThen: THEN WS rhsActionNoIfs (WS (addRemAction|outcomeConstraint|resetaction))*;
ifElse: (WS ELSE WS rhsActionNoIfs (WS (addRemAction|outcomeConstraint|resetaction))*)?;

//: addaction|remaction|termaction|passaction SEMICOLON;

// Support for if-then-else-endif statement
ifstatement: IF WS condition WS ifThen ifElse WS ENDIF;
condition: BRA WS? NOT? constraint (WS? andOR WS? constraint)* WS? KET;

resetaction: RESET WS alphanum;
termaction: TERMINATE WS? BRA outcome KET;
passaction: PASS;
addRemRopOperator: (ADDROP|REMROP);

addRemAction: alphanum DOT ropset WS? addRemRopOperator WS? upalphanum BRA alphanum
(COMMA timeSpec)? KET;

//addaction: alphanum DOT ropset WS? ADDROP WS? upalphanum BRA timeSpec? KET;
//remaction: alphanum DOT ropset WS? REMROP WS? upalphanum BRA timeSpec? KET;

// Rules for both lhs and rhs
outcome: SUCCESS | TECFAIL | INITFAIL | BIZFAIL;
ropset: RIGHTS|OBLIGS|PROHIBS;
timeSpec: absoluteTime; //|relativeTime; UNCOMMENT IF FIXED LATER
absoluteTime: DQUOTE DIGIT DIGIT DASH DIGIT DIGIT DASH DIGIT DIGIT DIGIT DIGIT
WS DIGIT DIGIT COLON DIGIT DIGIT COLON DIGIT DIGIT DQUOTE;
//relativeTime: relTimeElement+;

//relTimeElement: DIGIT+ ('s'|'m'|'h'|'d'|'M'|'Y');

// Token for declaration section
ROLEPLAYER: 'roleplayer';
BUSINESSOP: 'businessoperation';
COMPOBLIG: 'compoblig';

// Tokens for Basic rule structure
RULE: 'rule';
END: 'end';
WHEN: 'when';
THEN: 'then';

// Tokens for left hand side
MATCHES: 'matches';
HAPPENED: 'happened';
CTHAPPENED: 'counthappened';

```

BEFORE: 'before';
AFTER: 'after';
BOTYPE: 'botype';
ORIGINATOR: 'originator';
RESPONDER: 'responder';
OUTCOME: 'outcome';
TIMESTAMP: 'timestamp';
DAY: 'day';
DATE: 'date';
SECOND: 'second';
MINUTE: 'minute';
HOUR: 'hour';
MONTH: 'month';
YEAR: 'year';
IN: 'in';
NOTIN: '!in';
EQUALS: '==';
NOTEQ: '!=';
AND: '&&';
OR: '||';
NOT: '!';
WEEKDAY: 'Mon' | 'Tue' | 'Wed' | 'Thu' | 'Fri' | 'Sat' | 'Sun';
MONTHID: 'Jan' | 'Feb' | 'Mar' | 'Apr' | 'May' | 'Jun'
| 'Jul' | 'Aug' | 'Sep' | 'Oct' | 'Nov' | 'Dec';

// Tokens occurring in both lhs and rhs

SUCCESS: 'Success';
TECFAIL: 'TecFail';
BIZFAIL: 'BizFail';
INITFAIL: 'InitFail';
BOOLEANTRUE: 'true';
BOOLEANFALSE: 'false';

// Right hand side tokens

ADDROP: '+=';
REMROP: '-=';
TERMINATE: 'terminate';
PASS: 'pass';
RESET: 'reset';
OBLIGS: 'obligs';
RIGHTS: 'rights';
PROHIBS: 'prohibs';

// Tokens for Right hand side: structured statements

IF: 'if';
//THEN: 'then';
ELSE: 'else';


```

ENDIF: 'endif';

// Tokens for Right hand side: status guards
OTHERWISE: 'Otherwise';

// Identifiers, with uppercase and lowercase initials
upalphanum: UPPER (LOWER | UPPER | DIGIT)*;
alphanum: LOWER (LOWER | UPPER | DIGIT)*;
roleplayeridentiyer: alphanum; // change so that only alphanum is used.
bopIdentifier: upalphanum;
genericString: DQUOTE (LOWER | UPPER | DIGIT | WS | SEMICOLON
| COLON | COMMA | QUOTE | DOT | DASH | BACKSLASH)* DQUOTE;
// Alphabet, numbers
//LOWER: [a-z];
//UPPER: [A-Z];
//DIGIT: [0-9];
LOWER: 'a'..'z';
UPPER: 'A'..'Z';
DIGIT: '0'..'9';

// Various characters
SEMICOLON
: ';';
COLON
: ':';
HASH: '#';
BRA: '(';
KET: ')';
COMMA: ',';
QUOTE: '"';
DQUOTE: '\"'; // "
SQUAREBRA: '['; // may have to escape this
SQUAREKET: ']'; // may have to escape this
DOT: '.'; // may have to escape this
DASH: '-';
BACKSLASH: '\\';

WS : [ \t\r\n]+; // Define whitespace rule, toss it out

```